عبد اللطيف ايمش

تعلم JS
JavaScript

نظرة تفصيلية على الكائنات

تعلم JavaScript

ترجمة

عبد اللطيف ايمش

تقديم



رافقت زيادة استخدام شبكة الإنترنت زيادةً كبيرةً في الطلب على مطوري مواقع الويب، وتطوَّرت تقنيات الويب كثيرًا في الآونة الأخيرة، وأصبح تطوير واجهاتٍ تفاعليةً أمرًا هينًا يسهل القيام به لوجودِ كمٍّ كبيرٍ من المكتبات الجاهزة التي تعتمد على لغة JavaScript؛ لكن هذا أدى إلى عدم اهتمامِ جزءٍ كبيرٍ من المطورين باللغة الأساسية التي كُتِبَت فيها تلك المكتبات، مما يورث قصورًا في فهمهم لطريقة عمل تلك المكتبات، وكيفية تعاملها مع المتصفح خلف الستار.

لذا أتى هذا الكتاب محاولًا أن يشرح للمطوِّرين أصحاب المعرفة المتوسطة للغة JavaScript أو أولئك الذين يألفون استخدام مكتباتها كيف تعمل لغة JavaScript عبر شرحه للكائنات وما يتعلق بها شرحًا عميقًا يؤدي إلى فهم آلية عمل اللغة نفسها.

هذا الكتاب مترجمٌ عن كتاب «JavaScript Enlightenment» لصاحبه Cody Lindley، والذي نَشَرتَه دار نشر O'Reilly لاحقًا بنفس الاسم. نُشِرتَ هذه النسخة المترجمة بعد أخذ إذن المؤلف.

وفي النهاية، أحمد الله على توفيقه لي بإتمام العمل على الكتاب، وأرجو أن يكون إضافةً مفيدةً للمكتبة العربية، ويمكنكم التواصل معي على بريدي abdallatif.ey@gmail.com لأيّة استفسارات. والله ولي التوفيق.

عبد اللطيف محمد أديب ايمش
حلب، سورية 2017/1/15

هذا الكتاب برعاية



جدول المحتويات

الفصل الثاني: التعامل مع الكائنات والخاصيات.............77

الفصل الثالث: الكائن ()Object..............................103

الفصل الرابع: الكائن ()Function............................113

الفصل الخامس: الكائن الرئيسي العام....................139

الفصل السادس: الكلمة المحجوزة this....................147

الفصل التاسع: المصفوفات والكائن ()Array...............197

الفصل العاشر: السلاسل النصية ()String.................211

تمهيد



هذا الكتاب ليس عن أنماط التصميم في JavaScript ولا عن اتباع نموذج البرمجة كائنية التوجه في لغة JavaScript، وليس مكتوبًا للتعريف بالميزات الرائعة للغة JavaScript أو لتوضيح عيوبها، ولا يفترَض أن يكون مرجعًا شاملًا، وليس موجّهًا للأشخاص حديثي العهد بالبرمجة أو أولئك الذين لا يعرفون شيئًا عن JavaScript، وهذا ليس كتابًا يحتوي خطواتٍ لإنجاز أمرٍ معيّن؛ فجميع أنواع الكتب السابقة متوافرة من قبل.

غرضي من كتابة هذا الكتاب هو إعطاء القارئ نظرةً دقيقةً عن JavaScript من خلال استكشاف الكائنات في JavaScript وتوضيح الفروقات الدقيقة فيها مثل القيم المعقدة (complex values) والقيم الأولية (primitive values)، والمجال (scope)، والوراثة (inheritance)، والكائن الرئيسي (head object) ...إلخ. كان غرضي من هذا الكتاب أن يكون قصيرًا ويحتوي على خلاصةٍ مفهومةٍ للإصدار الثالث من مواصفة ECMA-262، وسأركّز فيه على طبيعة الكائنات في JavaScript.

إذا كنتَ مصممًا أو مطورًا استخدمتَ JavaScript من قبلُ تحت عباءة المكتبات (مثل jQuery، أو Prototype أو YUI ...إلخ.) فأرجو أن يحوّلك محتوى هذا الكتاب من مطوّرٍ يعتمدُ على مكتبة من مكتبات JavaScript إلى «مطوّر JavaScript».

1. لماذا كتبتُ هذا الكتاب؟

عليّ بدايةً أن أعترف أنّني كتبتُ هذا الكتاب لنفسي، كي أدوّن فيه معلوماتي ولا أعتمد تمامًا على ذاكرتي. إضافةً إلى الأسباب الآتية:

- تُسهّل المكتبات من حدوث متلازمة «الصندوق الأسود» التي قد تكون مفيدةً في بعض الأحيان لكن كارثيةً معظم الوقت! ستُنفَّذ الأمور بسرعة وكفاءة لكنك لا تملك أدنى فكرة

عن كيفية فعل ذلك أو لماذا. الحقيقة هي أنَّ أيَّ شخصٍ يخطط لاستخدام مكتبةٍ من مكتبات JavaScript أو إطارٍ عملٍ عندما يبني تطبيق ويب (أو عندما يُنشِئ نموذج تسجيل بسيط) يجب أن ينظر إلى ما وراء الستار ويفهم كيفية عمل محرِّك وأساس تلك المكتبة. هذا الكتاب مناسبٌ للذين يريدون أن يزيحوا الستار تمامًا ويكتبوا الشيفرات باستخدام JavaScript نفسها.

- توفِّر Mozilla دليلًا ومرجعًا كاملًا مُحدَّثًا لنسخة JavaScript 1.5؛ أعتقد أنَّ ما ينقصه هو مستندٌ مفهومٌ ومكتوبٌ من وجهةِ نظرٍ واحدةٍ يساعد في فهم المرجع المتوفر. أرجو أن يكون هذا الكتاب مَدخِلًا للمعلومات والمفاهيم غير المفصّلة في الدليل الذي توفره Mozilla.
- نسخة 1.5 من JavaScript ستتواجد لفترة طويلة، لكننا نتقدم تجاه إضافات جديدة في اللغة في الإصدارات التالية (مثل الإصدار السادس من ECMA)، لذا أردتُ أن أوثِّق المفاهيم الأساسية في JavaScript التي من غير المحتمل أن تتغير، وسأنوِّه ما استطعتُ إلى الاختلافات.
- الكتب التقنية المتقدمة التي تتحدث عن لغات البرمجة مليئةٌ بأمثلةٍ نظريةٍ عن الشيفرات وتحتوي على حشوٍ كثير. أنا أفضِّل وضع شرح قصير الذي يوصل القارئ إلى فهم الفكرة متبوعًا بمثالٍ عملي حقيقي الذي يمكن تشغيله مباشرةً. اتَّبعتُ في هذا الكتاب منهجًا يجزِّئ المواضيع المعقدة إلى مفاهيم أصغر وأقل تعقيدًا وأسهل استيعابًا تُشرَح بأقل قدرٍ ممكنٍ من الكلمات ومدعومةً بأمثلةٍ تفصيلية.
- ثخن أغلبية كتب JavaScript التي تستحق القراءة أكثر من 10 سم وأغلبها مراجع

تفصيلية لها مكانتها واستخداماتها؛ لكنني أردتُ إنشاء كتابٍ يضمُّ الأمور المهمة دون الاستفاضة كثيرًا.

2. من يجب عليه قراءة هذا الكتاب

هذا الكتاب موجهٌ إلى نوعين من الأشخاص. أول نوع هو المبتدئ الملم بالأساسيات أو مطوِّر JavaScript متوسط القدرات الذي يريد أن يقوي معرفته باللغة بفهمه لكائنات JavaScript فهمًا عميقًا. النوع الثاني هو الخبير في استخدام مكتبات JavaScript الذي أصبح جاهزًا للنظر إلى ما وراء الستار. هذا الكتاب ليس مناسبًا للوافدين الجدد على البرمجة أو على مكتبات JavaScript أو لِمَن أراد التعرّف على JavaScript.

3. لماذا إصدار JavaScript 1.5 (ECMA-262 v3)؟

سأركِّز في هذا الكتاب على إصدار 1.5 من JavaScript (الذي يكافئ ECMA-262 الإصدار الثالث) لأنَّ هذا الإصدار هو أكثر إصدار تطبيقًا في المتصفحات إلى حد الآن. سيُحدَّث الإصدار القادم من الكتاب لكي يحتوي على التحديثات والإضافات الموجودة في آخر إصدار من ECMA-262.

4. لماذا لم أشرح كائنات Date() و Error() و RegEx()؟

كما ذكرتُ سابقًا، هذا الكتاب ليس مرجعًا تفصيليًا للغة JavaScript، وإنما سيُركِّز على الكائنات التي ستساعدك على فهم JavaScript. لذا قررتُ ألّا أشرح الكائنات `Date()` و `Error()` و `RegEx()` (على الرغم من فائدتها الكبيرة) لأنَّ تعلّم تفاصيل هذه الكائنات لن يزيد أو ينقص من فهمك للكائنات في JavaScript. أرجو أن تُطبِّق ما ستتعلمه في هذا الكتاب على جميع

الكائنات الموجودة في بيئة JavaScript.

5. تنسيق الكتاب

قبل أن تبدأ، من المهم أن تفهم طريقة تنسيق الكتاب، رجاءً لا تتخطى هذا القسم لأنه يحتوي على معلوماتٍ مهمةٍ ستساعدك أثناء قراءتِكَ لهذا الكتاب.

أ. شيفراتٌ أكثر وكلماتٌ أقل

رجاءً تفحّص الشيفرات بدقة. يجب أن تنظر إلى الشرح كأمر ثانوي ملحقٌ بالشيفرة. شخصيًا أرى أنَّ الشيفرة تساوي ألف كلمة. لا تقلق إن زاد الشرح حيرتك في البداية، إذ عليك أن تتفحص الشيفرة وأن تقرأ التعليقات مرةً أخرى وتكرِّر هذه العملية إلى أن يصبح المفهوم أو الفكرة الذي أحاول شرحه واضحًا. أرجو أن تصل إلى مرحلةٍ من الخبرة لكيلا تحتاج إلا إلى شيفرةٍ موثقةٍ توثيقًا جيدًا لكي تستوعب أحد المفاهيم البرمجية.

ب. الكثير من الشيفرات والتكرار

ربما ستتضايق مني لتكرار نفس الأمور والاستفاضة في الأمثلة. ربما أستحق ذلك، لكنني أفضِّل أن أكون دقيقًا ومستفيضًا ومكرِّرًا، بدلًا من أن أضع اعتباراتٍ مغلوطةً عن القراء ومعلوماتهم، والتي يقع فيها المؤلفون عادةً. قد ترى أنَّ كلا الأمرين ممل، وذلك يعتمد على معرفتك بالموضوع، لكن لا تغفل أنَّ ذلك سيفيد الذين يحاولون تعلم موضوعٍ ما بالتفصيل.

ت. التنسيق والألوان

سأستخدمُ الخط العريض في شيفرات JavaScript (كما في المثال الآتي) للإشارة إلى الشيفرات والأسطر البرمجية التي تتعلق مباشرةً بالمفهوم الذي نشرحه، وسأستعمل اللون الفضي

الفاتح للإشارة إلى التعليقات:

```
<!DOCTYPE html><html lang="en"><body><script>

// هذا تعليقٌ عن الشيفرة لتوضيحها
var foo = 'calling out this part of the code';

</script></body></html>
```

بالإضافة إلى تنسيق الشيفرات، سأضيف في متن النص بعض شيفرات JavaScript، وتلك الشيفرات ستُنسَّق بخطٍ ذي عرضٍ ثابت بلونٍ فضيٍ غامق لكي تميّز بينها وبين النص العادي، مثال:

«ليكن لدينا كائن cody الذي أنشأناه من الدالة البانية `Object()` الذي لا يختلف عن كائنٍ لسلسلةٍ نصيةٍ أنشأناه باستخدام الدالة البانية `String()` على سبيل المثال. تفحص الشيفرة الآتية لتفهم ما أقصده (مثال حي):»

ث. التجربة الحية للأمثلة

أغلبية أمثلة هذا الكتاب مرتبطة بصفحة خاصة بها في jsFiddle (حيث أضع قبل الشيفرة رابطًا إليها بعنوان «مثال حي»)، حيث يمكنك تعديل وتنفيذ الشيفرة مباشرةً؛ أمثلة jsFiddle تستخدم إضافة Firebug lite-dev لذا ستعمل دالة عرض الناتج (أقصد `console.log`) في معظم المتصفحات الحديثة دون مشاكل. أرى أنَّ عليك أن تتعرف على الغرض من الدالة `console.log` وكيفية استخدامها قبل قراءة هذا الكتاب.

هنالك حالات يُسبِّب فيها موقع jsFiddle بعض التعقيدات مع شيفرة JavaScript، فعندها

سأستخدم JS Bin. حاولتُ أن أتفادى الاعتماد على المتصفح عبر استخدامي لإضافة Firebug lite-dev لكن في بعض الأحيـان يجب الاعتمـاد على المتصـفح في إظهـار المخرجـات، إن لم يملـك متصفحك console فأنصحك بترقيته إلى متصفحٍ حديث.

الفصل الأول:

الكائنات في JavaScript



1. إنشاء الكائنات

الكائنات هي اللبُنة الأساسية في JavaScript: إذ أنَّ كل شيءٍ فيها عبارةٌ عن كائنٍ أو يسلكُ سلوك كائن؛ لذا إذا فهمتَ الكائنات فستفهم JavaScript. لننظر إذًا إلى طريقة إنشاء الكائنات في JavaScript.

الكائن (object) هو مجرد حاوية أو مجموعة من القيم المرتبطة بأسماء (وتُسمى «الخاصيات» [properties]). لنحاول استيعاب ذلك منطقيًا أولًا قبل أن ننظر إلى أيّة شيفرات JavaScript. ما رأيك أن نأخذ شخصًا كمثال! يمكننا أن نصف الشخص «cody» باستخدام اللغة العربية في جدولٍ كالآتي:

cody	
الخاصية	**قيمة الخاصية**
على قيد الحياة (living)	true
العمر (age)	33
الجنس (gender)	ذكر (male)

الكلمة «cody» في الجدول السابق هي عنوانٌ لمجموعةٍ من «الخاصيات» و«القيم الموافقة لها» التي تُعرِّف ما هو «cody» تحديدًا. يمكنك أن تَعرِف من الجدول السابق أنَّ الشخص «cody» على قيد الحياة وعمره 33 سنة وهو ذكر.

لكن JavaScript لا تستعمل الجداول وإنما الكائنات، التي لن تختلف بُنيتها كثيرًا عن الجدول

السابق. يمكن تحويل المعلومات الموجودة في الجدول أعلاه إلى شيفرة JavaScript كالآتي (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// cody إنشاء الكائن
var cody = new Object();

// ثم ملء الكائن cody بالخاصيات
// (بفصل اسم الخاصية عن اسم الكائن بنقطة)
cody.living = true;
cody.age = 33;
cody.gender = 'male';

// الناتج: Object {living = true, age = 33, gender = 'male'}
console.log(cody);

</script></body></html>
```

أبقِ هذه المعلومة في ذهنك: الكائنات هي مجرد حاويات للخاصيات، وكل خاصية لها اسم وقيمة. تَستعمِل JavaScript مفهوم « الحاويةُ التي تضمُ خاصياتٍ فيها قيمٌ ذاتُ أسماءٍ» (أي «كائن») كحجر بناءٍ للتعبير عن القيم في JavaScript. الكائن cody هو قيمةٌ التي عَبَّرتُ عنها ككائن JavaScript بإنشائي لكائنٍ وإعطائِهِ اسمًا ومن ثم إعطاءُ قيمٍ لخاصياته.

في هذه المرحلة، الكائن cody الذي نناقشه لا يحتوي إلا على معلوماتٍ ثابتة؛ ولمّا كنّا نتعامل

مع لغةٍ برمجية، فمن المؤكد أنّنا نطمح لأن نبرمج الكائن cody لكي يفعل شيئًا ما. وإلا فكل ما نملكه هو مجرد قاعدةُ بياناتٍ ذات بنيةٍ قريبةٍ من صيغة JSON[1]. ولكي يفعل الكائن cody شيئًا ما، فيجب إضافة دالة (method) إليه، وتلك الدالة تقوم بوظيفة ما. ولكي نكون دقيقين، الدوال في JavaScript هي خاصياتٌ تحتوي على كائن `Function()` الذي يكون الهدف منه هو إجراء عمليةٍ على الكائن الذي يحتوي على الدالة.

إذا أردتُ تحديث جدول cody بإضافة دالة `getGender`، فسيبدو الجدول بالعربية كالآتي:

cody	
الخاصية	قيمة الخاصية
على قيد الحياة (living)	true
العمر (age)	33
الجنس (gender)	ذكر (male)
getGender	إعادة قيمة الجنس

باستخدام JavaScript، ستبدو دالة `getGender` الموجودة في الجدول المُحدَّث كالآتي (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>
```

1 اختصار للعبارة «JavaScript Object Notation» وهي صيغةٌ نصيةٌ تُستعمَل لتناقل البيانات، وهي سهلةُ القراءة والكتابة، تدعم تخزين مجموعة من الأزواج «الاسم/القيمة» أو قائمة من القيم المرتبة.

```
var cody = new Object();
cody.living = true;
cody.age = 33;
cody.gender = 'male';
cody.getGender = function(){return cody.gender;};

console.log(cody.getGender()); // الناتج: 'male'

</script></body></html>
```

تُستعمَل الدالة getGender -التي هي خاصيةٌ للكائن cody- لإعادة (return) قيمة من قيم إحدى خاصيات الكائن، التي هي القيمة «male» المُخزَّنة في الخاصية gender. ما عليك أن تفهمه أنَّه دون وجود دوال، فالكائن لا يفعل شيئًا سوى تخزين الخاصيات الثابتة.

الكائن cody الذي ناقشناه إلى الآن يُعرَف أيضًا ككائن من النوع ()Object، لقد أنشأنا كائن cody باستخدام كائن فارغ الذي وفرته لنا الدالة البانية (constructor function) لـ()Object. تخيل أنَّ الدوال البانية هي قوالبٌ لتوليد كائنات سَبَقَ تعريفها؛ وفي حالة كائن cody استخدمنا الدالة البانية ()Object لإنشاء كائن فارغ الذي أسميتُه cody، ولمّا كان cody كائنًا مبنيًا من الدالة البانية ()Object، فيمكننا أن نقول أنَّ cody هو كائنٌ من النوع ()Object. ما عليك فهمه -بغض النظر عن طريقة إنشاء كائنات بسيطة من النوع ()Object مثل cody- هو أنَّ غالبية القيم الموجودة في JavaScript هي كائنات (القيم الأولية مثل "foo" و 5 و true هي استثناء لهذه القاعدة، لكن توجد كائنات مكافئة لها، وسنتحدث في أمرها لاحقًا).

ليكن لدينا كائن cody الذي أنشأناه من الدالة البانية Object() الذي لا يختلف -على سبيل المثال- عن كائنٍ لسلسلةٍ نصيةٍ أنشأناه باستخدام الدالة البانية String(). تفحص الشيفرة الآتية لتفهم ما أقصده (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// Object() إنشاء كائن من نوع
var myObject = new Object();
myObject['0'] = 'f';
myObject['1'] = 'o';
myObject['2'] = 'o';

// الناتج: Object { 0="f", 1="o", 2="o"}
console.log(myObject);

// String() إنشاء كائن من نوع
var myString = new String('foo');

console.log(myString); // الناتج: foo { 0="f", 1="o", 2="o"}

</script></body></html>
```

كما يبدو، الكائنان myObject و myString هما... كائنان! يمكن أن يملك كلاهما خاصياتٍ أو يرث (inherit) خاصياتٍ، وتم إنشاؤهما من دالةٍ بانية. المتغير myString الذي يحتوي على قيمة السلسلة النصية «foo» يبدو بسيطًا جدًا، لكنه يملك بنية كائن وراء الستار. إذا تفحصت كلا

الكائنين اللذان أنُشِئا فستلاحظ أنها متماثلان بالمحتوى لكنهما مختلفان بالنوع. أهم ما في الأمر هو أن تلاحظ أنَّ JavaScript تستخدم الكائنات للتعبير عن القيم.

ملاحظة	ربما تجد من الغريب أن ترى القيمة «foo» على شكل كائن لأنَّ السلاسل النصية تُمثَّل في JavaScript على أنَّها قيم أولية (مثلًا `var myString = 'foo';`). استخدمتُ هنا كائن لإنشاء سلسلة نصية للفت انتباهك إلى أنَّ أيَّ شيءٍ يمكن أن يكون كائنًا، بما في ذلك القيم التي لا تفكر عادةً بها على أنها كائنات (أقصد السلاسل النصية، والأعداد، والقيم المنطقية [boolean]). وأظن أيضًا أنَّ هذا سيساعدك في فهم لماذا البعض يقول أنَّ كل شيءٍ في JavaScript هو كائن.

تحتوي لغة JavaScript على الدوال البانية `String()` و `Object()` في أساس اللغة لجعل عملية إنشاء كائن `String()` أو `Object()` بسيطةً. لكنك -كمطوِّر يستعمل لغة JavaScript- تستطيع أيضًا إنشاء دوال بانية تماثلها بالقوة. وضَّحتُ ذلك في المثال الآتي بتعريف دالة بانية خاصة باسم `Person()`، لذا سأتمكن من إنشاء «أشخاص» باستخدامها (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// تعريف دالة بانية باسم Person لاستعمالها لاحقًا
// لإنشاء كائنات Person()
var Person = function(living, age, gender) {
    this.living = living;
    this.age = age;
    this.gender = gender;
```

```
    this.getGender = function() {return this.gender;};
};

// Person إنشاء نسخة لكائن من الدالة البانية
// cody وتخزينها في المتغير
var cody = new Person(true, 33, 'male');

console.log(cody);

/*
الدالة البانية String() في الأسفل -المُعرّفة في أساس لغة
 Javascript- لها نفس طريقة الاستخدام. ولأنّ الدالة البانية
 لكائن السلسلة النصية من أساس لغة JavaScript، فكل ما علينا
 فعله للحصول على نسخة من الكائن لسلسلة نصية ما هو تهيئة
 الكائن. لكن طريقة الاستخدام هي نفسها سواءً استخدمنا الدوال
 البانية من أساس اللغة مثل String() أو استعملنا الدوال
 البانية التي عرفناها بأنفسنا مثل Person()
*/

// myString إنشاء نسخة من النوع String وتخزينها في المتغير
var myString = new String('foo');

console.log(myString);

</script></body></html>
```

يمكن للدالة البانية `Person()` التي عرفناها بأنفسها أن تُنشِئ كائناتٍ لأشخاص، كما تتمكن الدالة البانية `String()` من إنشاء كائناتٍ لسلاسلَ نصيةٍ. الدالة البانية `Person()` ليست أقلُّ قدرةً أو مرونةً من الدالة البانية `String()` أو غيرها من الدوال البانية الموجودة في أساس لغة JavaScript.

تذكَّر كيف أُنشِئ الكائن `cody` (الذي ناقشناه سابقًا) من الدالة البانية `Object()`. من المهم ملاحظة أنَّ الدالة البانية `Object()` والدالة البانية الجديدة `Person()` (الظاهرة في الشيفرة أعلاه) تنتجان نفس النتائج تمامًا. فكلتاهما تنتجان كائنًا له نفس الخاصيات ونفس الدوال. امعن النظر في قسمَي المثال الآتي، اللذان يظهران أنَّ للكائنين `codyA` و `codyB` نفس القيم، حتى لو أنشأناهما بطرائق مختلفة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// إنشاء كائن codyA باستخدام الدالة البانية Object()

var codyA = new Object();
codyA.living = true;
codyA.age = 33;
codyA.gender = 'male';
codyA.getGender = function() {return codyA.gender;};

// الناتج: Object {living=true, age=33, gender="male", …}
console.log(codyA);
```

```
/*
سننشئ نفس كائن cody هنا، لكن بدلًا من استخدام الدالة البانية
 Object() لإنشاء cody، فسنعرِّف أولًا الدالة البانية Person()
 التي ستمكننا من إنشاء الكائن cody (وأي كائن آخر نريده)؛
ومن ثم سنُنشِئ كائنًا عبر استدعائها باستخدام الكلمة المحجوزة
new
*/

var Person = function(living, age, gender) {
    this.living = living;
    this.age = age;
    this.gender = gender;
    this.getGender = function() {return this.gender;};
};

var codyB = new Person(true, 33, 'male');

// الناتج: Object {living=true, age=33, gender="male", …}
console.log(codyB);

</script></body></html>
```

الاختلافُ الرئيسيُ بين codyA و codyB لا يكمن في الكائن نفسه، وإنما في الدوال البانية المستخدمة لإنشاء الكائنات. أُنشِئ الكائن codyA باستخدام الدالة البانية Object(). أما الدالة البانية Person() فأنشأتْ الكائن codyB لكن يمكن استخدامها مرةً أخرى لتعريف كائنات جديدة

من النوع `Person()`. سيؤدي إنشاء دوال بانية خاصة بك لإنشاء كائنات أيضًا إلى ضبط وراثة تستعمل سلسلة prototype ‏(prototypal inheritance) لنسخ الكائن `Person()` (لا تقلق، سنُفصِّل ذلك لاحقًا).

سيؤدي كلا الحلين السابقين إلى إنشاء نفس الكائن المعقد؛ الطريقتان السابقتان هما أشهر الطرائق المستعملة لبناء الكائنات.

لغة JavaScript هي لغةٌ تأتي محملةً ببعض الدوال البانية في أساس اللغة والمستعملة لإنشاء كائنات معقدة التي تُعبِّر عن نوعٍ مُحدَّدٍ من القيم (مثلًا الأعداد، أو السلاسل النصية، أو الدوال، أو الكائنات، أو المصفوفات ...إلخ.)؛ بالإضافة إلى «المواد الخام» (كائنات `Function()`) التي يمكننا استعمالها لإنشاء دوال بانية خاصة بنا (مثلًا `Person()`) والنتيجة النهائية -بغض النظر عن النمط المستعمل لإنشاء الكائن- هي إنشاء كائن معقد.

ستُركِّز بقية الكتاب على فهم طريقة إنشاء الكائنات والقيم الأولية المكافئة لها، وطبيعتها، واستخدامها.

2. الدوال البانية في JavaScript تبني وتُعيد نسخًا من الكائن

دور الدالة البانية هو إنشاء عدّة كائنات التي تتشارك بخاصياتٍ وسلوك معيّن. بمفهومها المُبسَّط، الدالة البانية تشبه قالب تقطيع الكعكات لإنشاء كائنات لها خصائص افتراضية ودوال تابعة لها.

إذا قلتَ «الدالةُ البانيةُ ما هي إلا دالةٌ عاديةٌ»، فسأرد عليك قائلًا «أصبتَ، لكن تلك الدالة ستُستدعى باستخدام الكلمة المحجوزة new» (مثلًا `new String('foo')`)، وعندما يحدث ذلك، فستأخذ تلك الدالة دورًا خاصًا، وستُعامِل JavaScript تلك الدالة معاملةً خاصةً بضبط قيمة

الكلمة المحجوزة `this` لتلك الدالة إلى الكائن الجديد الذي سيُنشَأ. وبالإضافة إلى السلوك الخاص السابق، ستعيد هذه الدالةُ الكائنَ المُنشأَ حديثًا (أي `this`) افتراضيًا بدلًا من القيمة `false`. والكائن الجديد المُعاد من الدالة سيُعتبَر أنَّه نسخةٌ تابعةٌ للدالة البانية التي أنشأته.

خذ الدالة البانية `Person()` كمثال مرةً أخرى، لكن هذه المرة اقرأ التعليقات في الشيفرة الآتية بتمعّن، لأنها ستُبيّن لك تأثير الكلمة المحجوزة `new` (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

/*
 الدالة Person هي دالة بانية، وكُتِبَت لغرض استخدامها مع
الكلمة المحجوزة new
*/

var Person = function Person(living, age, gender) {
    // الكلمة المحجوزة this هي الكائن الجديد الذي سيُنشَأ
    // (this = new Object(); أي)
    this.living = living;
    this.age = age;
    this.gender = gender;
    this.getGender = function() {return this.gender;};
    // عندما تُستدعى هذه الدالة بالكلمة المحجوزة new
    // فستُعاد قيمة this بدلًا من false
};
```

```
// إنشاء نسخة من الكائن Person باسم cody
var cody = new Person(true, 33, 'male');

// cody هو كائن وهو نسخة من Person()
console.log(typeof cody); // الناتج: object

// الناتج هو الخاصيات الداخلية وقيمها التابعة للكائن
console.log(cody);
// الناتج هو إظهار بُنية الدالة البانية
console.log(cody.constructor);

</script></body></html>
```

الشيفرة السابقة تستعمل دالةً بانيةً مُعرِّفةً من قِبل المستخدم (أي `Person()`) لإنشاء الكائن cody. وهذا لا يختلف عن استعمال الدالة البانية `Array()` لإنشاء كائن من النوع `Array()` (مثلًا `new Array()`):

```
<!DOCTYPE html><html lang="en"><body><script>

// إنشاء نسخة من الكائن Array باسم myArray
var myArray = new Array();

// المتغير myArray هو كائنٌ منشأٌ من الدالة البانية Array()
// الناتج هو object! لا تستعجب، المصفوفات هي كائنات
console.log(typeof myArray);
```

```
console.log(myArray); // الناتج: [ ]

console.log(myArray.constructor); // الناتج: Array()

</script></body></html>
```

أغلبية القيم في JavaScript (ما عدا القيم الأولية) تتضمن إنشاء كائن (أو ما يسمى instantiation) من الدالة البانية. يسمى الكائن المُعاد من الدالة البانية «بالنسخة» (instance). خذ وقتك للتأقلم مع هذه الألفاظ والاصطلاحات، وكذلك طريقة استخدام الدوال البانية لإنشاء الكائنات.

3. الدوال البانية للكائنات الموجودة في أساس لغة JavaScript

تحتوي لغة JavaScript على تسع دوال بانية للكائنات موجودة في أساس اللغة (أو مُضمَّنة فيها). تُستعمَل هذه الكائنات من قِبل JavaScript لبناء اللغة، ومصطلح «البناء» أقصد فيه تلك الكائنات التي تُستخدم للتعبير عن قيم الكائنات في شيفرة JavaScript، وأيضًا لتوفير عدّة ميزات من ميزات اللغة. وبالتالي فإنَّ الدوال البانية للكائنات الموجودة في أساس لغة JavaScript هي دوالٌ متعددة الجوانب في أنَّها تستطيع إنشاء كائنات ولكن يمكن أيضًا استعمالها للمساعدة في إجراء الكثير من الأمور التي تفعلها لغات البرمجة. على سبيل المثال، الدوال هي كائناتٌ مُنشَأةٌ من الدالة البانية `Function()`، لكنها يمكن أن تُستعمَل أيضًا في إنشاء كائنات أخرى عندما تُستدعى كدالة بانية باستخدام الكلمة المحجوزة `new`.

هذه قائمة بتسع دوال بانية للكائنات التي تأتي مُضمَّنةً مع لغة JavaScript:

- `Number()`
- `String()`
- `Boolean()`
- `Object()`
- `Array()`
- `Function()`
- `Date()`
- `RegExp()`
- `Error()`

لغة JavaScript مبنية (تقريبًا) على الكائنات التسعة السابقة (بالإضافة إلى القيم الأولية التي هي السلاسل النصية والأعداد والقيم المنطقية [boolean]). فهم تلك الكائنات بالتفصيل هو المفتاح للاستفادة من القدرة البرمجية الاستثنائية للغة JavaScript وسيبرز لك مدى مرونة اللغة وكفاءتها.

ملاحظات	- الكائن Math هو كائن غريب بعض الشيء، إذ أنَّه كائنٌ «ساكن» (static) ، بدلًا من كونه دالةً بانيةً، وهذا يعني أنَّك لا تستطيع أن تكتب `var x = new Math()`، لكنك تستطيع استخدامه كما لو أنَّه مُهيّئ من قبل (مثلًا `Math.PI`). وفي الواقع، `Math` هو مجال أسماء (object namespace) مضبوطٌ من لغة JavaScript لتضمين الدوال التي تُعنى بالرياضيات. - الكائنات المُضمَّنة بلغة JavaScript يُشار إليها في بعض الأحيان «بالكائنات العامة» (global objects) لأنها كائناتٌ متاحةٌ للاستخدام ومضمّنةٌ في أساس لغة JavaScript. لكن لا يختلط عليك هذا المصطلح بمصطلح «الكائن الرئيسي

العام» (head global object) الذي هو الكائن الموجود في أعلى مستوى في سلسلة المجال (scope chain) ، مثلًا الكائن window الموجود في جميع متصفحات الويب. سنتحدث عن هذا الأمر بالتفصيل لاحقًا.

- الدوال البانية Number() و String() و Boolean() لا تبني الكائنات فحسب، وإنما توفِّر أيضًا قيمًا أوليةً للسلاسل النصية والأعداد والقيم المنطقية؛ وذلك اعتمادًا على طريقة استدعاء الدالة البانية. فلو استدعيتَ تلك الدوال البانية مباشرةً، فسيُعاد كائنٌ معقد؛ أما لو عبّرتَ ببساطة عن رقمٍ أو سلسلةٍ نصيةٍ أو قيمةٍ منطقيةٍ في الشيفرة (القيم الأولية مثل 5 و "foo" و true) فستُعيد الدالة البانية قيمةً أوليةً بدلًا من كائنٍ معقد.

4. الدوال البانية للكائنات التي يُنشِئها المستخدم

كما رأيت سابقًا عند إنشائنا لدالة Person()، من المسموح في JavaScript إنشاء الدوال البانية الخاصة بنا، التي يمكننا استخدامها لإنشاء أكثر من نسخة من الكائن.

سأريك في المثال الآتي دالةً بانيةً شبيهةً بدالة Person() السابقة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var Person = function(living, age, gender) {
    this.living = living;
    this.age = age;
    this.gender = gender;
    this.getGender = function() {
```

```
        return this.gender;
    };
};

var cody = new Person(true, 33, 'male');
// الناتج: Object {living=true, age=33, gender="male", …}
console.log(cody);

var lisa = new Person(true, 34, 'female');
// الناتج: Object {living=true, age=34, gender="female", …}
console.log(lisa);

</script></body></html>
```

كما تلاحظ، بتمرير معاملات (parameters) فريدة عند استدعاء الدالة البانية `Person()`، فستتمكن بسهولة من إنشاء عدد كبير من الكائنات الفريدة التابعة لأشخاص. يمكن أن تستفيد من هذا كثيرًا عندما تحتاج إلى إنشاء أكثر من ثلاثة كائنات لها نفس الخاصيات لكن بقيم مختلفة. هلّم لنتفكّر في الأمر، هذا ما تفعله JavaScript تمامًا عند تعاملها مع الكائنات المُضمَّنة فيها؛ فالدالة البانية `Person()` تتبع نفس المبادئ التي تتبعها الدالة البانية `Array()`، لذا لن تختلف `new Array('foo', 'bar')` كثيرًا عن `new Person(true, 33, 'male')`. إنشاء الدوال البانية الخاصة بك ما هو إلا اتباعك لنفس النمط الذي تستخدمه JavaScript للدوال البانية للكائنات المُضمَّنة فيها.

ملاحظات

- من المستحسن عند إنشاء دوال بانية ستُستخدَم مع الكلمة المحجوزة `new` أن نجعل الحرف الأول من اسم الدالة كبيرًا، مثلًا `Person()` بدلًا من `person()`؛ لكن ذلك ليس إجباريًا.

- أحد الأشياء التي عليك الانتباه إليها هو استخدام القيمة `this` داخل الدالة. تذكَّر أنَّ الدالة البانية ما هي إلا قالبٌ لتقسيم الكعكات؛ وعندما تستعملها مع الكلمة المحجوزة `new`، فستُنشِئ كائنًا يملك الخاصيات والقيم المُعرَّفة داخل الدالة البانية. وعندما نستعمل الكلمة المحجوزة `new`، فهذا يعني أنَّ الكلمة المحجوزة `this` تُشير إلى الكائن (أو النسخة) التي ستُنشأ بناءً على التعليمات البرمجية الموجودة داخل الدالة البانية. لكن على الجانب الآخر، إذا أنشأتَ دالةً بانيةً واستدعيتها دون استخدام الكلمة المحجوزة `new` فستشير قيمة `this` إلى الكائن «الأب» (parent) الذي يحتوي على الدالة. لا تقلق، ستُفصِّل ذلك باستفاضة في الفصل السادس.

- من الممكن أن نستغني عن استخدام الكلمة المحجوزة `new` ومفهوم الدوال البانية بإنشائنا لدالةٍ تُعيد كائنًا؛ لكن يجب أن تُكتَب هذه الدالة بطريقةٍ معينة لإنشاء كائن من نوع `Object()` وإعادته:

```
var myFunction = function() {return {prop : val}};
```

5. استدعاء الدوال البانية باستخدام المعامل new

الدالة البانية بأبسط مفهومٍ لها هي قالبٌ لتقطيع الكعكات يُستخدَم لإنشاء كائنات مُضبوطة مسبقًا. لنأخذ `String()` على سبيل المثال؛ هذه الدالة -عندما تُستعمَل مع المعامل (operator) `new` (أي `new String('foo')`)- ستُنشِئ سلسلةً نصيةً اعتمادًا على «القالب» `String()`.

لننظر إلى الشيفرة الآتية (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myString = new String('foo');

// الناتج: foo {0 = "f", 1 = "o", 2 = "o"}
console.log(myString);

</script></body></html>
```

أنشأنا أعلاه كائنًا نصيًّا من الدالة البانية `String()`. وبهذه الطريقة استطعنا أن نُعبِّر عن قيمةٍ نصيّة في JavaScript.

ملاحظة	أنا لا أقترح عليك استخدام الدوال البانية بدلًا من القيم الأولية المكافئة لها (مثل `var string="foo";`)، لكنني أرمي أن تفهم ما الذي يحدث في كواليس القيم الأولية.

وكما ذكرتُ سابقًا، تملك لغة JavaScript الدوال البانية التسع الآتية المُضمّنة في أساس اللغة:

`Number()` و `String()` و `Boolean()` و `Object()` و `Array()` و `Function()` و `Date()` و `RegExp()` و `Error()`. يمكننا أن نُنشِئ كائنًا من أيّة دالة بانية من الدوال السابقة باستخدام المعامل `new`. سأُنشِئ في المثال الآتي تسعة كائنات من الدوال المُضمَّنة في أساس اللغة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>
```

```
// new إنشاء نسخة من كل دالة بانية مُضمّنة باللغة باستخدام

var myNumber = new Number(23);
var myString = new String('male');
var myBoolean = new Boolean(false);
var myObject = new Object();
var myArray = new Array('foo','bar');
var myFunction = new Function("x", "y", "return x*y");
var myDate = new Date();
var myRegExp = new RegExp('\bt[a-z]+\b');
var myError = new Error('Crap!');

// إظهار أيّة دالة بانية قامت بإنشاء الكائن
console.log(myNumber.constructor); // الناتج: Number()
console.log(myString.constructor); // الناتج: String()
console.log(myBoolean.constructor); // الناتج: Boolean()
console.log(myObject.constructor); // الناتج: Object()
console.log(myArray.constructor); //الناتج: Array()
console.log(myFunction.constructor); // الناتج: Function()
console.log(myDate.constructor); // الناتج: Date()
console.log(myRegExp.constructor); // الناتج: RegExp()
console.log(myError.constructor); // الناتج: Error()

</script></body></html>
```

عند استخدامنا للمعامل new، فإننا نُخبِر مفسِّر JavaScript أننا نريد إنشاء كائن من الدالة البانية الموافقة لنوعه. على سبيل المثال، الدالة البانية `Date()` تُستعمَل لإنشاء كائنات الوقت والتاريخ. الدالة البانية `Date()` هي مجرد «قالب» لكائنات الوقت والتاريخ. وهذا يعني أنَّها تنتج كائنات باستخدام النمط الافتراضي المُعرَّف من الدالة البانية `Date()`.

يجب أن تكون الآن مستوعبًا لطريقة إنشاء نسخ لكائنات من الدوال البانية مضمَّنة باللغة (مثل `new String('foo')`) ومن الدوال البانية التي عرَّفها المستخدم (مثل `new Person (true, 33, 'male')`).

ملاحظة	أبقِ بذهنك أنَّ `Math` هو كائن ساكن (static object) -أي أنَّه حاوية لدوال أخرى- وليس له دالة بانية التي تَستعمِل معها المعامل new.

6. الطرائق المختصرة لإنشاء القيم من الدوال البانية

توفِّر JavaScript طرائق مختصرة -نسميها «literals»- لإنشاء أغلبية قيم الكائنات المُضمَّنة فيها دون الحاجة إلى استخدام `new Foo()` أو `new Bar()`. يعطي الشكل المختصر نفس الكائن الذي كان سيُنشَأ إذا استخدمنا المعامل new. الاستثناءات من هذه القاعدة هي: `Number()` و `String()` و `Boolean()` (انظر الملاحظة في الأسفل).

إذا كانت لديك خلفية برمجية، فربما ستكون معتادًا على استخدام الطريقة المختصرة لإنشاء الكائنات. سأنشِئ في المثال الآتي كائنات باستدعاء الدالة البانية باستخدام المعامل new ومن ثم سأُنشِئ كائنات مكافئة لها باستخدام الطريقة المختصرة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myNumber = new Number(23); // كائن
var myNumberLiteral = 23; // قيمة عددية أوليّة، وليست كائنًا

var myString = new String('male'); // كائن
// قيمة نصيّة أوليّة، وليست كائنًا //
var myStringLiteral = 'male';

var myBoolean = new Boolean(false); // كائن
// قيمة منطقية أوليّة، وليست كائنًا //
var myBooleanLiteral = false;

var myObject = new Object();
var myObjectLiteral = {};

var myArray = new Array('foo', 'bar');
var myArrayLiteral = ['foo', 'bar'];

var myFunction = new Function("x", "y", "return x*y");
var myFunctionLiteral = function(x, y) {return x*y};

var myRegExp = new RegExp('\bt[a-z]+\b');
var myRegExpLiteral = /\bt[a-z]+\b/;

// تبيين أنّ الكائنات المُنشَأة من الطريقة المختصرة //
```

```
// ستُنشَأ من نفس الدالة البانية

console.log(myNumber.constructor,
myNumberLiteral.constructor);
console.log(myString.constructor,
myStringLiteral.constructor);
console.log(myBoolean.constructor,
myBooleanLiteral.constructor);
console.log(myObject.constructor,
myObjectLiteral.constructor);
console.log(myArray.constructor, myArrayLiteral.constructor);
console.log(myFunction.constructor,
myFunctionLiteral.constructor);
console.log(myRegExp.constructor,
myRegExpLiteral.constructor);

</script></body></html>
```

ما الذي عليك أن تعيه هنا هو أنَّ الطريقة المختصرة تُبسّط وتخفي عملية إنشاء الكائنات مقارنةً باستخدام المعامل new. ربما تراها أنَّها أكثر وضوحًا وأسهل قراءةً.

حسنًا، أصبحت الأمور أكثر تعقيدًا بخصوص القيم الأولية للسلاسل النصية والأعداد والقيم المنطقية. في تلك الحالات ستأخذ القيم المُنشَأة بالطريقة المختصرة خصائص القيم الأولية بدلًا من قيم الكائنات المعقدة. انظر الملاحظة في الأسفل للتفاصيل.

ملاحظة	عند استخدام الطريقة المبسطة لإنشاء القيم النصية أو العددية أو المنطقية، فلن يُنشَأ كائنٌ معقدٌ حتى تُعامَل القيمة ككائن. بعبارةٍ أخرى، ستتعامل مع نوعٍ أوليٍّ من القيم إلى أن تحاول استخدام دوال أو الحصول على خاصيات مرتبطة بالدالة البانية (مثلًا `var charactersInFoo = 'foo'.length`)، وعندما يحدث ذلك، فستُنشِئ JavaScript كائنًا لاحتواء القيمة الأوليّة في الكواليس، مما يتيح لك أن تعامِل القيمة ككائن. ثم بعد استدعائك للدالة، فستُهمِل JavaScript الكائن وستعود القيمة إلى قيمةٍ أوليّة. وهذا هو السبب وراء تسمية السلاسل النصية والأعداد والقيم المنطقية بأنها أنواع بيانات «أوليّة» أو «بسيطة». أرجو أن يوضِّح ما سبق اللبس والخلط الناتج عن تداخل المفهوم «كل شيء في JavaScript عبارةٌ عن كائنٍ» مع المفهوم «كل شيء في JavaScript يمكن أن يَسلِكَ سلوكَ كائنٍ».

7. القيم الأولية (أو البسيطة)

تُعتبَر القيم 5 و `'foo'` و `ture` و `false` وأيضًا `null` و `undefined` في JavaScript على أنَّها قيمٌ أوليةٌ لأنها غيرُ قابلةٍ للاختزال. أي أنَّ العدد هو مجموعة من الأرقام، والسلسلة النصية هي مجموعة من المحارف، والقيم المنطقية إما أن تكون `true` أو `false`، والقيم `null` و `undefined` هي مجرد `null` أو `undefined`! هذه القيم بسيطة بطبيعتها، ولا تُمثِّل قيمًا يمكن أن تتألف أو تتكوّن من قيمٍ أخرى.

تفحّص الشيفرة الآتية واسأل نفسك إذا كانت السلاسل النصية أو الأعداد أو القيم المنطقية أو `null` أو `undefined` يمكن أن تصبح أكثر تعقيدًا؛ ثم قارن ذلك بما تعرفه عن نسخةٍ من النوع `Object()` أو `Array()` أو أيُّ كائنٍ معقّدٍ آخر (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myString = 'string';
var myNumber = 10;
// false أو ture يمكن أن تكون
var myBoolean = false;
var myNull = null;
var myUndefined = undefined;

console.log(myString, myNumber, myBoolean, myNull,
myUndefined);

/*
تخيّل أنّ كائنًا معقدًا مثل المصفوفات يمكن أن يتكون من عدِّة قيم
أولـيّة، وهذا يعني أنّه سيصبح مجموعة معقدة مكوّنة من عدِّة قيم.
*/

var myObject = {
    myString: 'string',
    myNumber: 10,
    myBoolean: false,
    myNull: null,
    myUndefined: undefined
};

console.log(myObject);
```

```
var myArray = ['string', 10, false, null, undefined];

console.log(myArray);

</script></body></html>
```

الأمر بسيطٌ جدًا: القيم الأوليّة تُمثِّل أبسطَ شكلٍ من المعلومات أو المعطيات المتاحة في لغة JavaScript.

ملاحظات	- على عكس إنشاء الكائنات بالشكل المبسّط: عند إنشاء قيمة `String()` أو `Number()` أو `Boolean()` باستخدام الكلمة المحجوزة `new` فعندئذٍ سيُنشأ كائنٌ معقد. - من المهم جدًا أن تفهم أنَّ الدوال البانية للكائنات `String()` و `Number()` و `Boolean()` هي دوالٌ بانيةٌ ثنائيةُ الغرضِ التي يمكن أن تُستعمَل لإنشاءِ قيمٍ أوليّةٍ بالإضافةِ إلى القيم المُعقَّدة. هذه الدوال البانية لا تُعيد كائناتٍ دائمًا، وإنما تستطيع أن تُعيدَ تمثيلًا أوليًا لقيمة الكائن المعقد إن استدعِيَت دون استخدام المعامل `new`.

8. القيم الأولية null و undefined و "string" و 10 و true و false ليست كائنات

القيم null و undefined هي قيمٌ بسيطةٌ جدًا والتي لا تحتاج أصلًا إلى دالة بانية أو إلى

استخدام المعامل new لاستعمالها كقيمة في JavaScript. لاستخدام null أو undefined فكل ما ستحتاج إليه هو استعمالها كما لو كانت مُعامِلًا (operator). القيم الأولية الباقية (السلاسل النصية، والأعداد، والقيم المنطقية) ليست كائنات على الرغم من أنَّها تُعاد من دالةٍ بانيةٍ.

سأوضِّح في المثال الآتي الفرق بين القيم الأولية وبقية كائنات JavaScript المُضمَّنة فيها (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// لن تُنشَأ أيّة كائنات عند استخدام القيم الأولية
// لاحظ عدم استخدام الكلمة المحجوزة new
var primitiveString1 = "foo";
var primitiveString2 = String('foo');
var primitiveNumber1 = 10;
var primitiveNumber2 = Number('10');
var primitiveBoolean1 = true;
var primitiveBoolean2 = Boolean('true');

// التأكد (من ناتج typeof) أنّ القيمة الأولية ليست كائنًا
console.log(typeof primitiveString1, typeof
primitiveString2); // الناتج: 'string,string'
console.log(typeof primitiveNumber1, typeof
primitiveNumber2); // الناتج: 'number,number'
console.log(typeof primitiveBoolean1, typeof
primitiveBoolean2); // الناتج: 'boolean,boolean'
```

```
// لو استخدمنا دالةً بانيةً عبر المعامل new لإنشاء الكائنات

var myNumber = new Number(23);
var myString = new String('male');
var myBoolean = new Boolean(false);
var myObject = new Object();
var myArray = new Array('foo', 'bar');
var myFunction = new Function("x", "y", "return x * y");
var myDate = new Date();
var myRegExp = new RegExp('\\bt[a-z]+\\b');
var myError = new Error('Crap!');

// الناتج: 'object object object object object function
object function object'
console.log(
typeof myNumber,
typeof myString,
typeof myBoolean,
typeof myObject,
typeof myArray,
/*
انتبه أنّ المعامل typeof سيُعيد function لجميع كائنات الدوال
*/
typeof myFunction,
```

```
typeof myDate,
/*
انتبه أنّ المعامل typeof سيُعيد object لجميع كائنات RegExp()
*/
typeof myRegExp,
typeof myError
);

</script></body></html>
```

الذي أريد منك أن تفهمه من الشيفرة السابقة هي أنَّ القيم الأولية ليست كائنات؛ وإنما القيم الأولية ذات خصوصيةٍ في أنَّها تُستعمَل لتمثيل أبسط نوع من أنواع القيم.

9. كيف تُخزَّن وتُنسخ القيم الأولية في JavaScript

من المهم جدًا أن تفهم أنَّ القيم الأولية تُخزَّن وتُعالَج «كقيم اسمية» (face value)؛ وهذا يعني أنَّك إذا خزَّنتَ القيم "foo" في متغيرٍ باسم myString فستُخزَّن القيمة "foo" كما هي حرفيًا في الذاكرة. لكن قد تتساءل لماذا الأمر مهمٌ لهذه الدرجة؟ لأنَّك عندما تبدأ بمعالجة القيم (أي أن تنسخها مثلًا)، فيجب أن يكون هذا المفهوم مألوفًا لديك، لأنَّ القيم الأوليّة تُنسَخ حرفيًا.

ستُخزِّن في المثال التالي نسخةٌ من قيمة المتغير myString (ألا وهي 'foo') في المتغير myStringCopy، وستُنسَخ القيمة حرفيًا؛ وحتى لو عدَّلنا في القيمة الأصلية فستحافظ القيمة المنسوخة (المُشار إليها عبر المتغير myStringCopy) على قيمتها دون تغيير (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// إنشاء متغير يحتوي على قيمة نصيّة أولية
var myString = 'foo';
var myStringCopy = myString; // نسخ قيمته إلى متغيرٍ جديد

// تعديل القيمة المُخزّنة في المتغير myString
var myString = null;

/*
نُسِخَت القيمة الأصلية للمتغير myString إلى المتغير
myStringCopy. يمكن التأكد من ذلك بتعديل قيمة المتغير
myString ثم التحقق من قيمة المتغير myStringCopy.
*/

console.log(myString, myStringCopy); // الناتج: 'null foo'

</script></body></html>
```

الفكرة التي أريد إيصالها هنا هي أنَّ القيم الأولية تُخزَّن وتُعالَج كقيم غير قابلة للاختزال؛ والإشارة إليها تؤدي إلى نقل قيمتها. ففي المثال السابق، نسخنا قيمة المتغير `myString` إلى المتغير `myStringCopy`. ثم حدَّثنا قيمة `myString`، وبقي المتغير `myStringCopy` حاويًا على نسخةٍ من قيمة `myString` القديمة. تذكّر هذه الآلية المُتبَعة في نسخ القيم الأولية وقارنها مع الكائنات المعقدة (التي سنشرحها لاحقًا).

10. القيم الأولية تتساوى اعتمادًا على القيمة

يمكن مقارنة القيم الأولية لمعرفة إن تساوت قيمها حرفيًا. وكما هو واضح منطقيًا، إذا قارنتَ متغيرًا يحتوي العدد 10 بمتغيرٍ آخر يحتوي العدد 10، فستَعتبر JavaScript أنَّ المتغيرين متساويان لأنَّ 10 تساوي 10 تمامًا (أي 10 === 10 وهذا أمرٌ بدهي). والمِثل ينطبق فيما إذا قارنا السلسلة النصية الأوليّة 'foo' بسلسلةٍ نصيةٍ أوليةٍ أخرى لها القيمة 'foo' أيضًا. ناتج عملية المقارنة هو المساواة لأنَّ كلا السلسلتين متساويتين في القيمة (أي 'foo' === 'foo').

في الشيفرة الآتية، سأشرح مفهوم «المساواة اعتمادًا على القيمة» باستخدام القيم الأوليّة وسأقارن ذلك بكائنٍ معقد (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var price1 = 10;
var price2 = 10;
// كائن عددي معقد بسبب استخدامنا للمعامل new
var price3 = new Number('10');
var price4 = price3;

console.log(price1 === price2); // الناتج: true

/*
الناتج false لأنّ price3 يحتوي على كائن عددي معقد و price1 هو
قيمة أوليّة.
*/
```

```
console.log(price1 === price3);

/*
 الناتج true لأنّ القيم المعقدة تتساوى بالمرجعية (reference)
 وليس بالقيمة.
*/
console.log(price4 === price3);

/*
ماذا لو غيرنا المتغير price4 ليحتوي على قيمة أوليّة؟
*/
price4 = 10;

/*
الناتج false لأنّ price4 أصبح قيمةً أوليّة بدلًا من كائنٍ معقد.
*/
console.log(price4 === price3);

</script></body></html>
```

الفكرة هنا هي أنَّه عند مقارنة القيم الأولية فسيتم التحقق من أنَّ «القيم» متساوية. أما عندما نُنشِئ سلسلةً نصيةً أو عددًا أو قيمةً منطقيةً باستخدام الكلمة المحجوزة new (مثلًا new Number('10')) فلن تبقى القيمةُ أوليّةً. ولهذا السبب لن تكون نتيجة المقارنة مماثلةً للنتيجة التي نحصل عليها عندما نقارن القيم الأولية. وهذا ليس أمرًا يُستعجب منه، بعد الأخذ بعين الاعتبار أنَّ القيم الأولية ستُخزَّن بقيمتها (أي 10 === 10) بينما تُخزَّن القيم المعقدة

بمرجعيتها (reference) أي «هل يحتوي المتغيران `price3` و `price4` على مرجع لنفس القيمة».

11. القيم النصية والعددية والمنطقية الأولية ستسلك سلوك كائن عندما نعاملها ككائنات

عندما نُعامِل القيم الأولية كما لو أنّها كائنٌ مُنشَأٌ من دالةٍ بانية، فستحوّلها JavaScript إلى كائنٍ لكي تستطيع إجراء العملية المُحدَّدة عليها، لكنها -أي JavaScript- ستُهمِل بعدئذٍ الكائنَ الذي أنشأته وتعود إلى القيمة الأوليّة. سأُنشِئ في المثال الآتي قيمًا أوليّةً وأريك ما سيحدث عندما تُعامَل القيم الأولية ككائنات (مثال حي[2]):

```
<!DOCTYPE html><html lang="en"><body><script>

// إنشاء قيم أوليّة
var myNull = null;
var myUndefined = undefined;
var primitiveString1 = "foo";
// لم نستخدم هنا المعامل new، لذا سنحصل على قيمة أوليّة
var primitiveString2 = String('foo');

var primitiveNumber1 = 10;
// لم نستخدم هنا المعامل new، لذا سنحصل على قيمة أوليّة
var primitiveNumber2 = Number('10');
```

2 عُدِّل المثال قليلًا لتبيان نوع المتغير

```
var primitiveBoolean1 = true;
// لم نستخدم هنا المعامل new، لذا سنحصل على قيمة أوليّة //
var primitiveBoolean2 = Boolean('true');

/*
محاولة الوصول إلى الدالة toString() (الموروثة من
 object.prototype) لتوضيح كيف ستُحوّل القيم إلى كائنات عندما
تُعامل ككائنات.
*/

// الناتج: "foo" "foo"
console.log(primitiveString1.toString(),
primitiveString2.toString());
// الناتج: "string" "string"
console.log(typeof primitiveString1, typeof
primitiveString2);

// الناتج: "10" "10"
console.log(primitiveNumber1.toString(),
primitiveNumber2.toString());
// الناتج: "number" "number"
console.log(typeof primitiveNumber1, typeof
primitiveNumber2);

// الناتج: "true" "true"
```

```
console.log(primitiveBoolean1.toString(),
primitiveBoolean2.toString());
//الناتج: "boolean" "boolean"
console.log(typeof primitiveBoolean1, typeof
primitiveBoolean2);

/*
سيظهر خطأ هنا، لأنّ null و undefined لا يمكن تحويلهما إلى
كائنات، ولا يملكان دالةً بانيةً.
*/
console.log(myNull.toString());
console.log(myUndefined.toString());

</script></body></html>
```

في الشيفرة السابقة، كل القيم الأولية (ما عدا null و undefined) تم تحويلها إلى كائنات، وذلك لكي تستطيع استخدام الدالة ()toString عليها، ثم ستعود إلى أصلها كقيم أوليّة بعد أن ينتهي تنفيذ الدالة.

12. القيم المُعقَّدة (أو المركبة)

الدوال البانية للكائنات المُضمَّنة في JavaScript (أي ()Object و ()Array و ()Function و ()Date و ()Error و ()RegExp) هي قيمٌ معقدة (complex) وذلك لأنها تحتوي على قيمة أوليّة أو معقدة (أو أكثر من قيمة واحدة). بشكلٍ أساسي، تتكون القيم المعقدة من مختلف أنواع الكائنات في JavaScript؛ ويمكننا القول أنَّ القيم المعقدة لا تملك حجمًا تخزينيًا

معروفًا في الذاكرة لأنّ الكائنات المعقدة يمكن أن تحتوي على أيّ قيمة دون تحديد نوع معيّن من القيم. سنُنشِئ في المثال الآتي كائنًا ومصفوفةً يحتوي كلٌ منهما على القيم الأولية (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var object = {
    myString: 'string',
    myNumber: 10,
    myBoolean: false,
    myNull: null,
    myUndefined: undefined
};

var array = ['string', 10, false, null, undefined];

/*
 قارن ما سبق ببساطة القيم الأولية الآتية. إذ لا يمكن أن تكون
 أيّة قيمة من القيم أدناه أكثر تعقيدًا مما هي عليه، بينما
 يمكن أن تتضمن القيم المعقدة على أيّة أنواع من القيم
الموجودة في JavaScript (أنظر الشيفرة في الأعلى).
*/

var myString = 'string';
var myNumber = 10;
var myBoolean = false;
var myNull = null;
```

```
var myUndefined = undefined;

</script></body></html>
```

الفكرة التي أريد إيصالها هنا أنَّ القيم المعقدة هي مجموعة من القيم والتي تختلف بتعقيدها وبنيتها عن القيم الأولية.

ملاحظة	المصطلح «كائن معقد» (complex object) يُعبَّر عنه في كتبٍ أخرى بمصطلح «كائنات مركبة» (composite objects) أو «أنواع مرجعية» (reference types). إن لم ترَ أنَّ تلك المصطلحات بدهية، فأحب أنَّ أوضِّح أنها تصف طبيعة القيم في JavaScript باستثناء القيم الأولية؛ إذ أنَّ القيم الأولية ليست ذات «مرجعية بالقيمة» (reference by value) ولا يمكن أن تُمثِّل شيئًا مركبًا (أي شيءٌ ما مكوّنٌ من عدّة أجزاء أو عناصر) من قيمٍ أخرى. بينما الكائنات المعقدة لها «مرجعية بالقيمة» (referenced by value) ويمكن أن تحتوي على القيم الأخرى.

13. كيف تُخزَّن أو تُنسَخ القيم المعقدة في JavaScript

من المهم جدًا أن تفهم أنَّ القيم المعقدة تُخزَّن وتعالَج بالمرجعية (by reference). فعندما تُنشِئ متغيرًا يحتوي على كائنٍ معقد، فإن القيمة ستُخزَّن في الذاكرة في عنوان (address). فعندما تُشير إلى كائنٍ معقد، فإنك تستخدم اسمه (أي: أحد المتغيرات أو إحدى خاصيات الكائن) للحصول على القيمة الموجودة في ذاك العنوان في الذاكرة. ما سبق سيؤثر كثيرًا عندما تفكر ما الذي سيحدث لو حاولتَ نسخ قيمة معقدة. سأنشِئ في المثال التالي كائنًا مُخزَّنًا في المتغير

myObject، ثم ستُنسَخ قيمة المتغير myObject إلى المتغير copyOfMyObject؛ وستلاحظ أنَّ المتغير copyOfMyObject في الحقيقة ليس نسخةً من الكائن وإنما نسخةٌ من عنوان الذاكرة الذي يحتوي على الكائن (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject = {};

// لم تُنسَخ القيمة، وإنما نُسِخَت المرجعية (reference) فقط
var copyOfMyObject = myObject;

// تعديل القيمة المُخزّنة في الكائن myObject
myObject.foo = 'bar';

/*
إذا عرضنا الآن المتغيرين myObject و copyOfMyObject، فستلاحظ
أنّهما يملكان نفس القيمة للخاصية foo، لأنها يُشيران إلى نفس
الكائن.
*/

// الناتج: 'Object { foo="bar"} Object { foo="bar"}'
console.log(myObject, copyOfMyObject);

</script></body></html>
```

ما عليك أن تدركه هو أنَّ الكائنات (أي القيم المعقدة) -على عكس القيم الأولية التي تنسخ

قيمتها- تُخزَّن بالمرجعية. أي أنَّ المرجعية (أو العنوان) سيُنسَخ، لكن لن تُنسَخ القيمة الفعلية. وهذا يعني أنَّ الكائنات لا تُنسَخ بتاتًا. وكما قلتُ سابقًا، الذي سيُنسَخ هو العنوان أو المرجع إلى الكائن في الذاكرة. وفي مثالنا السابق أشار المتغيران `myObject` و `copyOfMyObject` إلى نفس الكائن في الذاكرة.

الفكرة المهمة هنا هي أنَّك عندما تُغيّر قيمةً معقدةً فستُغيّر القيمة المخزنة في جميع المتغيرات التي تُشير بالمرجعية إلى تلك القيمة المعقدة. ففي الشيفرة السابقة، ستتغير قيمة المتغيرين `myObject` و `copyOfMyObject` في كل مرة تُحدِّث فيها محتوى أحدهما.

ملاحظات	
	- عندما تستعمل كائنًا من النوع `String()` أو `Number()` أو `Boolean()` أنشأته باستخدام الكلمة المحجوزة `new`، أو حولته إلى كائن مركّب في الكواليس، فستُخزَّن وتُنسَخ القيم المخزنة في تلك الكائنات كما في القيم الأولية. إذًا، حتى لو كان بالإمكان معاملة القيم الأولية كقيم مركبة، لكنها لن تماثلها في جزئية النسخ بالمرجعية. - لإنشاء نسخة فعلية من كائنٍ ما، فيجب عليك استخلاص القيم يدويًا من الكائن القديم ووضعها في الكائن الجديد. يجدر بالذكر أنَّ الإصدار السادس من ECMAScript يحتوي على الدالة `Object.assign()` التي يمكن أن تُستخدَم لنسخ الكائنات، راجع صفحة الدليل لمزيدٍ من المعلومات.

14. الكائنات المعقدة تتساوى اعتمادًا على المرجعية

عندما نقارن الكائنات المعقدة، فسنعلم أنَّها تتساوى عندما تُشير إلى نفس الكائن (أي أنَّ لها نفس العنوان في الذاكرة). فإذا حوى متغيران كائنين متماثلين، فلن يكونا متساويين لأنهما لا

يشيران إلى نفس الكائن.

في المثـال أدنـاه لـدينا objectFoo و objectBar يملكـان نفس الخاصـيات وهمـا -في الواقع- كائنان متماثلان تمامًا. لكن عندما نقارنهما باستخدام === فستخبرنا JavaScript أنَّهما غير متماثلين (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var objectFoo = {same: 'same'};
var objectBar = {same: 'same'};

// false الناتج هو
// لأنّ JS لا تلقي بالًا لتماثل الكائنات بالقيم أو بالنوع
console.log(objectFoo === objectBar);

// المتغيران لهما نفس المرجعية، وهما «مساويان» لبعضهما
var objectA = {foo: 'bar'};
var objectB = objectA;

// الناتج هو true، لأن المتغيرين يشيران إلى نفس الكائن
console.log(objectA === objectB);

</script></body></html>
```

الفكرة هنا هي أنَّ المتغيرات التي تُشير إلى كائنٍ معقد في الذاكرة تكون متساويةً إذا كان لها نفس «العنوان». والعكس صحيحٌ أيضًا، لا يمكن أن يتساوى كائنان مستقلان حتى لو كان لهما

نفس النوع واحتويا على نفس الخاصيات.

15. للكائنات المعقدة خاصياتٌ ديناميكية

المتغير الجديد الذي يُشير إلى كائنٍ معقدٍ موجودٍ مسبقًا لا يؤدي إلى نسخ الكائن، وهذا هو السبب وراء تسمية الكائنات المعقدة بالكائنات المرجعية. فيمكن أن يكون للكائن المعقد أيُّ عددٍ تريده من المرجعيات، والتي ستشير جميعها إلى نفس الكائن، حتى لو تغيّر ذاك الكائن (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var objA = {property: 'value'};
var pointer1 = objA;
var pointer2 = pointer1;

// حدِّث خاصية objA.property، وستُحدّث جميع المرجعيات
// (أي pointer1 و pointer2)
objA.property = null;

// الناتج هو 'null null null'
// لأنّ objA و pointer1 و pointer2 تُشير إلى نفس الكائن
console.log(objA.property, pointer1.property,
pointer2.property);

</script></body></html>
```

وهذا يسمح لنا بالحصول على خاصيات ديناميكية للكائنات لأنَّك تستطيع تعريف كائن، ومن

ثم إنشاء مرجعيات له، ثم تحديث كائن، و«ستُحدَّث» كل المتغيرات التي تُشير إلى ذاك الكائن.

16. المعامل typeof يُستعمل على القيم الأولية والمعقدة

يمكن أن يُستعمَل المعامل typeof لإعادة نوع القيمة التي تتعامل معها؛ لكن القيم المُعادة منه قد لا تتوافق مع القيم التي تتوقعها منطقيًا؛ سأريك ذلك في المثال الآتي الذي يُظهِر القيم المعادة من استخدام المعامل typeof على مختلف أنواع القيم (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// قيم أولية
var myNull = null;
var myUndefined = undefined;
var primitiveString1 = "string";
var primitiveString2 = String('string');
var primitiveNumber1 = 10;
var primitiveNumber2 = Number('10');
var primitiveBoolean1 = true;
var primitiveBoolean2 = Boolean('true');

// الناتج هو كائن؟ ماذا؟! انتبه له…
console.log(typeof myNull);
// الناتج هو undefined
console.log(typeof myUndefined);
// الناتج هو string string
console.log(typeof primitiveString1, typeof
```

```
primitiveString2);
// number number الناتج هو
console.log(typeof primitiveNumber1, typeof
primitiveNumber2);
// boolean boolean الناتج هو
console.log(typeof primitiveBoolean1, typeof
primitiveBoolean2);

// القيم المعقدة
var myNumber = new Number(23);
var myString = new String('male');
var myBoolean = new Boolean(false);
var myObject = new Object();
var myArray = new Array('foo', 'bar');
var myFunction = new Function("x", "y", "return x * y");
var myDate = new Date();
var myRegExp = new RegExp('\\bt[a-z]+\\b');
var myError = new Error('Crap!');

// object :الناتج
console.log(typeof myNumber);
// object :الناتج
console.log(typeof myString);
// object :الناتج
console.log(typeof myBoolean);
// object :الناتج
```

```
console.log(typeof myObject);
// object :الناتج
console.log(typeof myArray);
// الناتج هو function؟ ماذا؟! انتبه لهذا…
console.log(typeof myFunction);
// object :الناتج
console.log(typeof myDate);
// object :الناتج
console.log(typeof myRegExp);
// object :الناتج
console.log(typeof myError);

</script></body></html>
```

عندما تَستعمِل هذا المعامل على القيم، فيجب أن تكون ملمًا بالناتج الذي من المحتمل إعادته لمختلف أنواع القيم (سواءً كانت أوليةً أم معقدةً) التي تتعامل معها.

17. الخاصيات الديناميكية تسمح بتغيير الكائنات

الكائنات المعقدة تتألف من خاصياتٍ ديناميكيةٍ وهذا يسمح بتغيير الكائنات التي يُعرِّفها المستخدم بالإضافة إلى أغلبية الكائنات المُضمَّنة في لغة JavaScript. وهذا يعني أنَّ غالبية الكائنات في JavaScript يمكن أن تُحدَّث أو تُغيّر في أيّ وقت. وبسبب ذلك، يمكننا أن نُغيّر الطبيعة الأساسية المضبوطة مسبقًا للغة JavaScript نفسها بتعديلنا وتغييرنا لكائناتها المُضمَّنة فيها. هذا لا يعني أنَّني أريد منك فعل ذلك، وفي الواقع أرى أنَّ عليك اجتنابه. لكن دعنا لا نجعل آرائنا تتدخل بما هو ممكن.

هذا يعني أنَّ بالإمكان تخزين خاصيات في الدوال البانية الأساسية المُضمَّنة في اللغة وإضافة دوال جديدة إلى الكائنات الأساسية مع إضافات إلى كائنات prototype.

في الشيفرة الآتية، سأغيّر في الدالة البانية ()String وفي كائن String.prototype (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

/*
الزيادة على الدالة البانية String عبر خاصية
augmentedProperties
*/
String.augmentedProperties = [];
// إذا لم يحتوي الكائن prototype على دالة trimIT() فأضفها
if (!String.prototype.trimIT) {
    String.prototype.trimIT = function() {
        return this.replace(/^\s+|\s+$/g, '');
    }
    /*
أضف الآن السلسلة النصية trimIT إلى المصفوفة
    augmentedProperties
    */
    String.augmentedProperties.push('trimIT');
}
var myString = '  trim me  ';
```

```
// النـاتج: 'trim me'
console.log(myString.trimIT());

// النـاتج: 'trimIT'
console.log(String.augmentedProperties.join());

</script></body></html>
```

مـا أريـد إيضـاحه هـو أنَّ الكائنـات في JavaScript ديناميكيـة، وهـذا يسـمح للكائنـات في JavaScript أن تُعدَّل. وهذا يعني أيضًا أنَّ بإمكانك تغيير اللغة كلها إلى نسخةٍ معدَّلة (إضافة الدالة `trimIT` كمثال). لكنني أكرِّر أنَّي لا أستحسن ذلك، وإنما ذكرتُه هنا لكي تعرفَ أنَّ هذا جزءٌ من طبيعة الكائنات في JavaScript.

تحذير	احذر! إذا غيرتَ في طريقة العمل الداخلية للغة JavaScript، فمكن المرجح أنَّك ستحصل على نسخةٍ مخصصةٍ من اللغة. لكن كن حذرًا من فعل ذلك وأدرك أنَّ أغلبية الأشخاص سيعتبرون أنَّ JavaScript متماثلة في أي مكان تتوافر فيه.

18. جميع الكائنات تملك خاصية constructor التي تُشير إلى الدالة البانية لها

عند إنشاء أي كائن، فستُنشَأ أيضًا الخاصية `constructor` وراء الكواليس كخاصية لذاك الكائن (أو النسخة)؛ والتي تُشير إلى الدالة البانية التي أَنشَأت الكائن. أنشأنا في المثال الآتي كائنًا

من نوع `Object()` مخزنًا في المتغيّر foo، ومن ثم تحققنا أنَّ خاصية constructor متوافرة للكائن الذي أنشأناه (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = {};

// foo الناتج true، لأنّ الدالة Object() أنشأت الكائن
console.log(foo.constructor === Object);
// Object() ستتم الإشارة إلى الدالة البانية
console.log(foo.constructor);

</script></body></html>
```

يمكن أن أستفيد من هذه الخاصية إن كنتُ أعمل على نسخةٍ من كائنٍ ولا أعرف ما أو مَن الذي أنشأه (خصوصًا عندما أتعامل مع شيفرةٍ كتبها غيري)، وبهذه الطريقة سأتمكن من تحديد إن كان الكائنُ مصفوفةً أو تعبيرًا نمطيًا أو خلاف ذلك.

يمكنك أن تلاحظ في المثال الآتي أنَّني أنشأتُ نسخًا من أغلبية الكائنات التي تأتي مع لغة JavaScript. لاحظ أنَّ استخدام القيم الأولية لا يعني عدم إمكانية تبيان قيمة الخاصية constructor عندما تُعامل القيمة الأولية ككائن (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>
```

```
var myNumber = new Number('23');
// الطريقة المختصرة
var myNumberL = 23;

var myString = new String('male');
// الطريقة المختصرة
var myStringL = 'male';

var myBoolean = new Boolean('true');
// الطريقة المختصرة
var myBooleanL = true;

var myObject = new Object();
// الطريقة المختصرة
var myObjectL = {};

var myArray = new Array();
// الطريقة المختصرة
var myArrayL = [];

var myFunction = new Function();
// الطريقة المختصرة
var myFunctionL = function() {};

var myDate = new Date();
```

```
var myRegExp = new RegExp('/./');
// الطريقة المختصرة
var myRegExpL = /./;

var myError = new Error();

// true جميع التعابير الآتية ستُعيد
console.log(
    myNumber.constructor === Number,
    myNumberL.constructor === Number,
    myString.constructor === String,
    myStringL.constructor === String,
    myBoolean.constructor === Boolean,
    myBooleanL.constructor === Boolean,
    myObject.constructor === Object,
    myObjectL.constructor === Object,
    myArray.constructor === Array,
    myArrayL.constructor === Array,
    myFunction.constructor === Function,
    myFunctionL.constructor === Function,
    myDate.constructor === Date,
    myRegExp.constructor === RegExp,
    myRegExpL.constructor === RegExp,
    myError.constructor === Error
);
```

```
</script></body></html>
```

تعمل خاصية constructor أيضًا على الدوال البانية التي يُعرِّفها المستخدم. سنُعرِّف في المثال الآتي الدالة البانية CustomConstructor() ثم سنُنشِئ كائنًا عبر هذه الدالة باستعمالنا للمعامل new؛ وبعد أن نُنشِئ الكائن سنتمكن من الوصول إلى الخاصية constructor (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var CustomConstructor = function CustomConstructor(){ return
'Wow!'; };
var instanceOfCustomObject = new CustomConstructor();

// الناتج: true
console.log(instanceOfCustomObject.constructor ===
CustomConstructor);

// ستعيد مرجعًا إلى الدالة CustomConstructor()
// الناتج: 'function() { return 'Wow!'; };'
console.log(instanceOfCustomObject.constructor);

</script></body></html>
```

ملاحظات	- ربما تحتار لماذا توجد للقيم الأولية خاصية constructor التي تُشير إلى الدوال البانية بينما لا تُعيد تلك الدوال كائنًا. السبب هو أنَّه بالرغم من إعادة قيمة أولية، لكن الدالة البانية ستستدعى، لذا هنالك علاقة بين القيم الأولية والدوال البانية؛ على الرغم من أنَّ النتيجة النهائية هي قيمة أوليّة. - إذا أردتَ أنَّ يُسجّل الاسم الحقيقي للدالة البانية المُعرَّفة من قِبل المستخدم، فعليك إعطاء الدالة البانية اسمًا حقيقيًا (مثلًا var Person = (function Person(){};.

19. التحقق فيما إذا كان كائنٌ ما مُنشَأً من دالةٍ بانيةٍ معيّنة

باستخدام المعامل instanceof، يمكننا أن نُحدِّد (إما true أو false) إذا كان الكائن منشأً من دالةٍ بانيةٍ معيّنة.

سنتحقق في المثال الآتي إذا كان الكائن instanceOfCustomObject مبنيًّا من الدالة البانية CustomConstructor. هذا المعامل صالحٌ للاستخدام مع الكائنات التي أنشأها المستخدم والكائنات المُضمَّنة في اللغة والمُنشَأة بالمعامل new (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// دالة بانية مُعرفّة من قِبل المستخدم
var CustomConstructor = function() {this.foo = 'bar';};

// إنشاء كائن باستخدام الدالة البانية CustomConstructor
var instanceOfCustomObject = new CustomConstructor();
```

```
console.log(instanceOfCustomObject instanceof
CustomConstructor); // الناتج: true

// تعمل بنفس آلية الكائنات العادية
// الناتج: true
console.log(new Array('foo') instanceof Array);

</script></body></html>
```

ملاحظات

- هنالك أمرٌ عليك الاحتياط منه عند استعمالك للمعامل instanceof هو أنَّه سيُعيد true في كل مرة تستخدمه فيها للتحقق من أنَّ الكائن هو نسخة من Object لأنَّ كل الكائنات تنحدر من الدالة البانية ()Object.

- سيُعيد المعامِل instanceof القيمة false عندما تتعامل مع القيم الأولية التي يمكن أن تتحول إلى كائنات (مثلًا 'foo' instanceof String الناتج false). أما لو كانت لدينا السلسلة النصية 'foo' المُنشَأة باستعمال المعامل new مع الدالة البانية، فسيعيد المعامل instanceof القيمة true. لذا أبقِ ببالك أنَّ instanceof يعمل فقط مع الكائنات المعقدة والكائنات المُنشَأة من الدوال البانية التي تعيد كائنات.

20. يمكن أن يملك كائن مُنشأ من دالة بانية خاصياته المستقلة

يمكن أن تتغــــير (augmented) الكائنـــــات في JavaScript في أيّ وقت (أي أنَّ الخاصــــيات ديناميكيـة). وكما ذكرنا سابقًا أنَّ الكائنات في JavaScript قابلة للتغيير. وهذا يعني أنَّ الكائنات

التي أنُشِئت من دالة بانية يمكن أن تتغير خصائصها.

سأُنشِئ في المثال الآتي كائنًا من الدالة البانية `Array()` ثم سأغيّره ليملك خاصية مستقلة تابعة له (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = new Array();
myArray.prop = 'test';

console.log(myArray.prop) // الناتج: 'test'

</script></body></html>
```

يمكن فعل ذلك أيضًا مع الكائن `Object()` أو `RegExp()` أو أيّة دوال بانية أخرى لا تُعيد قيمًا أوليّة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// يمكن فعل هذا مع أيّة دالة بانية مُضمّنة باللغة وتعيد كائنًا
var myString = new String();
var myNumber = new Number();
var myBoolean = new Boolean(true);
var myObject = new Object();
var myArray = new Array();
```

```
var myFunction = new Function('return 2+2');
var myRegExp = new RegExp('\bt[a-z]+\b');

myString.prop = 'test';
myNumber.prop = 'test';
myBoolean.prop = 'test';
myObject.prop = 'test';
myArray.prop = 'test';
myFunction.prop = 'test';
myRegExp.prop = 'test';

// الناتج:
// 'test', 'test', 'test', 'test', 'test', 'test', 'test'
console.log(myString.prop,myNumber.prop,myBoolean.prop,myObje
ct.prop,myArray.prop,myFunction.prop, myRegExp.prop);

// انتبه: لا يمكن إنشاء خاصيات للقيم الأوليّة
var myString = 'string';
var myNumber = 1;
var myBoolean = true;

myString.prop = true;
myNumber.prop = true;
myBoolean.prop = true;

// الناتج: undefined, undefined, undefined
```

```
console.log(myString.prop, myNumber.prop, myBoolean.prop);

</script></body></html>
```

إضافة خاصيات إلى الكائنات المُنشَأة من الدوال البانية هو أمرٌ شائع. تذكَّر أنَّ الكائنات المُنشَأة من الدوال البانية ما تزال «كائنات»!

ملاحظة	ابقِ ببالك أنَّ الكائنات -بغض النظر عن الخاصيات التابعة لها- يمكن أن تملك خاصيات موروثة عبر سلسلة prototype (prototype chain)، أو يمكن إضافتها إلى الدالة البانية بعد التهيئة. وهذا يبيّن لك البنية الديناميكية للكائنات في JavaScript.

21. الاختلافات بين «كائنات JavaScript» و «كائنات Object()»

لا ترتبك ولا تخلط بين الاصطلاح «كائنات JavaScript» الذي يشير إلى مجموعة الكائنات الموجودة في JavaScript، مع كائنات `Object()` (مثلًا `var myObject = new Object()`) التي هي نوعٌ خاصٌ من القيم الموجودة في JavaScript. كما أنَّ كائن `Array()` هو نوعٌ من الكائنات التي تسمى array، فكائن `Object()` هو نوعٌ من الكائنات التي تسمى object. الخلاصة هي أنَّ الدالة البانية `Object()` تُنتِج حاويةً عموميةً للكائنات، والتي يُشار إليها بكائنات `Object()`. وبشكلٍ شبيه، الدالة البانية `Array()` تُنتِج كائن array، ونشير إلى تلك الكائنات بكائنات `Array()`.

سنستخدم الاصطلاح «كائنات JavaScript» في هذا الكتاب للإشارة إلى جميع الكائنات في

JavaScript، لأنَّ أغلبيـة القيم في JavaScript تسـلك سـلوك الكائنـات. وهـذا بسـبب أنَّ أغلبيـة القيم في JavaScript تُنشَأ من دالة بانية مُضمَّنة باللغة التي تُنتِج نوعًا محدَّدًا من الكائنات.

ما الذي عليك تذكره هو أنَّ كائن `Object()` هو نوعٌ خاصّ جدًا من القيم؛ إذ أنَّه كائنٌ عموميٌّ فـارغٌ، ولا يختلـط عليـك مع اصـطلاح «كائنـات JavaScript» المُسـتخدم للإشـارة إلى أغلبيـة القيم التي يمكن التعبير عنها في JavaScript ككائنات.

الفصل الثاني:

التعامل مع الكائنات والخاصيات

2

1. يمكن أن تحتوي الكائنات المعقدة على غالبية أنواع القيم في JavaScript كخاصيات

يمكن أن يحتوي الكائن المعقد على أيّة قيمة مسموحة في JavaScript. أنُشأتُ في المثال الآتي كائن `Object()` باسم `myObject` ومن ثم أضفتُ إليه خاصياتٍ تُمثِّل أغلبية القيم المتوافرة في JavaScript (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject = {};

// إنشاء قيم تابعة للكائن myObject التي تمثل أغلبية القيم
// الموجودة في JavaScript ضمنيًا

myObject.myFunction = function() {};
myObject.myArray = [];
myObject.myString = 'string';
myObject.myNumber = 33;
myObject.myDate = new Date();
myObject.myRegExp = /a/;
myObject.myNull = null;
myObject.myUndefined = undefined;
myObject.myObject = {};
myObject.myMath_PI = Math.PI;
myObject.myError = new Error('Crap!');
```

```
console.log
(myObject.myFunction,myObject.myArray,myObject.myString,myObj
ect.myNumber,myObject.myDate,myObject.myRegExp,myObject.myNul
l,myObject.myNull,myObject.myUndefined,myObject.myObject,myOb
ject.myMath_PI,myObject.myError);

// يمكن فعل المِثل لأي نوع من الكائنات المعقدة، مثل الدوال

var myFunction = function() {};

myFunction.myFunction = function() {};
myFunction.myArray = [];
myFunction.myString = 'string';
myFunction.myNumber = 33;
myFunction.myDate = new Date();
myFunction.myRegExp = /a/;
myFunction.myNull = null;
myFunction.myUndefined = undefined;
myFunction.myObject = {};
myFunction.myMath_PI = Math.PI;
myFunction.myError = new Error('Crap!');

console.log
(myFunction.myFunction,myFunction.myArray,myFunction.myString
,myFunction.myNumber,myFunction.myDate,myFunction.myRegExp,my
```

```
Function.myNull,myFunction.myNull,myFunction.myUndefined,myFu
nction.myObject,myFunction.myMath_PI,myFunction.myError);

</script></body></html>
```

ما وددتُ إيضاحه هنا هو أنَّ الكائنات يمكن أن تحتوي -أو تشير إلى- أي شيء يمكنك التعبير عنه في JavaScript. يجب ألّا تصاب بالدهشة لأنك رأيت ما سبق، لأنَّك تعرف أنَّ جميع الكائنات المُضمَّنة في اللغة يمكن تغييرها؛ وهذا ينطبق أيضًا على القيم `String()` و `Number()` و `Boolean()` المُعادة على شكل كائنات (أي أنَّها أُنشِئت باستخدام المعامل `new`).

2. تغليف الكائنات المعقدة بطريقة نستفيد منها برمجيًا

يمكن أن تحتوي الكائنات `Object()` و `Array()` و `Function()` على كائنات معقدة أخرى. سأشرح ما سبق بضبط شجرة من الكائنات باستعمال كائنات `Object()` (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// التغليف باستخدام الكائنات، مما ينشِئ سلسلةً من الكائنات
var object1 = {
    object1_1: {
        object1_1_1: {foo: 'bar'},
        object1_1_2: {}
    },
    object1_2: {
        object1_2_1: {},
```

```
        object1_2_2: {}
    }
};

// الناتج: 'bar'
console.log(object1.object1_1.object1_1_1.foo);

</script></body></html>
```

يمكن فعل المِثل باستخدام الكائن `Array()` (أي سينتج عندما مصفوفة متعددة الأبعاد)، أو مع كائن `Function()` (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// عند تغليف المصفوفات، سنحصل على مصفوفة متعددة الأبعاد
// مصفوفة فارغة، داخلها مصفوفة فارغة، داخلها مصفوفة فارغة
var myArray = [[[]]];

// هذا مثال عن التغليف باستخدام الدوال
// سننشئ دالة فارغة داخلها دالة فارغة داخل دالة فارغة
var myFunction = function() {
    // فارغة
    var myFunction = function() {
        // فارغة
        var myFunction = function() {
```

```
            // فارغة
        };
    };
};

// يمكننا إنشاء كائنات معقدة فيها أكثر من نوع
var foo = [{
    foo: [{
        bar: {
            say: function() {
                return 'hi';
            }
        }}]}];
console.log(foo[0].foo[0].bar.say()); // الناتج: 'hi'

</script></body></html>
```

الفكرة الأساسية هنا هي أنَّ بعض الكائنات المعقدة مصممة لكي تُغلِّف (encapsulate) الكائنات الأخرى بطرائقَ مفيدةٍ برمجيًا.

3. ضبط أو تحديث أو الحصول على قيمة خاصية من خاصيات الكائن باستخدام طريقة النقط أو الأقواس

يمكنك أن تضبط أو تحدث أو تحصل على قيمة خاصية من خاصيات الكائن إما عبر طريقة النقط (dot notation) أو طريقة الأقواس (bracket notation).

سأوضِّح في المثال الآتي طريقة النقط، التي تتلخّص باستعمال اسم الكائن متبوعًا بنقطة ثم اسم الخاصية التي نريد ضبط قيمتها أو تحديثها أو الحصول عليها (مثلًا objectName.property) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// cody إنشاء كائن من نوع Object() في المتغير
var cody = new Object();

// ضبط خاصياته
cody.living = true;
cody.age = 33;
cody.gender = 'male';
cody.getGender = function() {return cody.gender;};

// الحصول على قيم الخاصيات
console.log(
    cody.living,
    cody.age,
    cody.gender,
    cody.getGender()
); // الناتج: 'true 33 male male'

// تحديث الخاصيات يُطابق طريقة ضبطها
cody.living = false;
```

```
cody.age = 99;
cody.gender = 'female';
cody.getGender = function() {return 'Gender = ' +
cody.gender;};

console.log(cody);

</script></body></html>
```

طريقة النقط هي أشهر طريقة لضبط أو تحديث أو الحصول على قيمة لخاصيات كائنٍ ما.

أمَّا طريقة الأقواس -التي ما لم تكن ضروريةً- فهي أقل استعمالًا سأستعمل في المثال الآتي طريقة الأقواس بدلًا من طريقة النقط. حيث سيُتبَع اسم الكائن بقوس مربع للاستهلال (أي [) ثم اسم الخاصية (بين علامتَي اقتباس) ثم قوس مربع للإغلاق (أي]) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// cody في المتغير Object() إنشاء كائن من نوع
var cody = new Object();

// ضبط خاصياته
cody['living'] = true;
cody['age'] = 33;
cody['gender'] = 'male';
cody['getGender'] = function() {return cody.gender;};
```

```
// الحصول على قيم الخاصيات
console.log(
    cody['living'],
    cody['age'],
    cody['gender'],
    cody['getGender']() //  ضع أقواس استدعاء الدالة في
النهاية
); // الناتج: 'true 33 male male'

// تحديث الخاصيات شبيه جدًا بطريقة ضبطها
cody['living'] = false;
cody['age'] = 99;
cody['gender'] = 'female';
cody['getGender'] = function() {return 'Gender = ' +
cody.gender;};

console.log(cody);

</script></body></html>
```

يمكن أن تستفيد خير استفادة من طريقة الأقواس إذا احتجتَ إلى الوصول إلى مفتاح خاصية وعليك أن تتعامل مع متغيرٍ يحتوي على قيمة نصية تُمثِّل اسم الخاصية. سأشرح ميزة استخدام طريقة الأقواس على طريقة النقط في المثال الآتي بالوصول إلى الخاصية foobar، وذلك باستخدام متغيرين اللذان إذا جُمِعا فستنتج سلسلةٌ نصيةٌ تحتوي على اسم الخاصية

الموجودة في الكائن foobarObject (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foobarObject = {foobar: 'foobar'};

var string1 = 'foo';
var string2 = 'bar';

// لن نستطيع استخدام طريقة النقط لهذا الغرض
console.log(foobarObject[string1 + string2]);

</script></body></html>
```

إضافةً إلى ذلك، يمكن أن نستفيد من طريقة الأقواس للوصول إلى أسماء خاصيات التي تكون غير صالحة كمُعرِّفات في JavaScript. سأستخدم في المثال الآتي رقمًا وكلمةً محجوزةً كأسماءٍ للخاصيات (يسمح باستخدامها كسلسلة نصية) التي يمكن استخدام طريقة الأقواس فقط للوصول إليها (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject = {'123':'zero','class':'foo'};

// لا يمكننا استخدام طريقة النقط لفعل هذا!
// أبقِ ببالك أنّ class هي كلمة محجوزة في JavaScript
```

```
// الناتج: 'zero foo'
console.log(myObject['123'], myObject['class']);

// إذا حاولنا استخدام طريقة النقط، فسنحصل على خطأ
// console.log(myObject.0, myObject.class);

</script></body></html>
```

ملاحظات

- لمّا كان من الممكن أن تحتوي الكائنات على كائناتٍ أخرى، فليس من غير الشائع أن نرى cody.object.object.object أو cody['object']['object']['object']['object']. وهذا يسمى بسلسلة الكائنات (object chain). لا يوجد حد لمدى تغليف الكائنات.

- الكائنات في JavaScript قابلة للتغيير، وهذا يعني أنَّ ضبط أو تحديث أو الحصول على خاصياتها يمكن أن يتم على أغلبية الكائنات في أيّ وقت. وعند استخدامنا لطريقة الأقواس (مثلًا cody['age'])، فيمكننا أن نحاكي المصفوفات الترابطية (Associative Arrays) الموجودة في لغات البرمجة الأخرى.

- إن كانت خاصيةٌ ما داخل أحد الكائنات دالةً، فكل ما عليك فعله هو وضع المعاملين () أمامها (مثلًا cody.getGender()) لاستدعاء تلك الدالة.

4. حذف خاصيات الكائنات

يمكن أن يُستعمَل المعامل delete لحذف الخاصيات حذفًا كاملًا من الكائن. حذفنا في

المثال الآتي الخاصية bar من الكائن foo (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = {bar: 'bar'};
delete foo.bar;
// الناتج false، لأننا حذفنا bar من foo
console.log('bar' in foo);

</script></body></html>
```

ملاحظات

- المعامل delete لن يحذف الخاصيات الموجودة في سلسلة prototype.

- استخدام المعامل delete هو الطريقة الوحيدة لإزالة خاصية من الكائن، أما ضبط قيمة الخاصية إلى undefined أو null سيُغيّر من قيمة الخاصية، ولن يؤدي إلى حذف الخاصية نفسها.

5. كيفية استبيان الإشارات إلى خاصيات الكائن

إذا حاولتَ الوصول إلى خاصية غير موجودة في كائنٍ ما، فستحاول JavaScript أن تعثر على الخاصية أو الدالة باستخدام سلسلة prototype. سأنشِئ في المثال الآتي مصفوفةً وسأحاول الوصول إلى خاصيةٍ باسم foo التي لم تُعرَّف بعد. ربما ستظن أنَّه لما كانت الخاصية myArray.foo ليست من خاصيات الكائن myArray، فستُعيد JavaScript القيمة undefined مباشرةً؛ لكن JavaScript ستبحث في مكانَين آخرين (Array.protoype ومن ثم

Object.prototype) عن القيمة foo قبل أن تُعيد undefined (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = [];

console.log(myArray.foo); // الناتج: undefined

/*
ستبحث JS في Array.prototype عن الخاصية Array.prototype.foo
لكنها لن تجدها؛ ثم ستبحث عنها في Object.prototype ولن تجدها
أيضًا؛ ثم بعدئذٍ ستُعيد undefined.
*/

</script></body></html>
```

عندما أحاول الوصول إلى خاصية أحد الكائنات، فستتحقق JavaScript من قيمة الخاصية التابعة لتلك النسخة من الكائن. فإن امتلك الكائنُ الخاصيةَ فستعيدُ JavaScript قيمة تلك الخاصية، ولن تحدث عملية وراثة (inheritance) لأنَّ سلسلة prototype لم تُستعمَل. أما إذا لم تحتوي نسخة الكائن على تلك الخاصية، فستبحث JavaScript عنها في كائن prototype التابع للدالة البانية للكائن.

تملك جميع نسخ الكائنات خاصيةً تمثِّل رابطًا سريًا (أقصد __proto__) للدالة البناية التي أَنشأَت النسخة. يمكن أن يُستعمَل الرابط السري للحصول على الدالة البانية، تحديدًا «خاصية prototype» للدالة البانية لنسخة الكائن.

هذا أحد أكثر الجوانب المحيّرة في JavaScript، لذا دعنا نتمعّن جيدًا في الموضوع. تذكَّر أنَّ أيّة دالة هي عبارةٌ عن كائنٍ له خاصيات. ومن المنطقي السماح للكائنات بوراثة (inherit) الخاصيات من الكائنات الأخرى، كما لو قلنا «أهلًا أيها الكائن B، أريد منك أن تتشارك جميع الخاصيات التي يملكها الكائن A معه»؛ وتَفعل JavaScript هذا لجميع الكائنات المُضمَّنة فيها باستخدام الكائن `prototype`. وتستطيع استعمال سلسلة prototype أيضًا عندما تُنشِئ دوالًا بانيةً خاصةً بك.

ستبقى طريقة تنفيذ JavaScript لذلك غامضةً إلى أن ترى كيف أنَّها مجموعةٌ من القواعد فحسب. لنُنشِئ مصفوفةً لكي نتفحص الخاصية `prototype` عن قرب (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// Array المتغير myArray يحتوي على كائنٍ من نوع
var myArray = ['foo', 'bar'];

// Array.prototype.join الدالة join() مُعرّفة في
console.log(myArray.join());

</script></body></html>
```

النسخة التي أنشأناها من `Array()` هي كائنٌ يملك خاصياتٍ ودوالًا، وعندما نحاول استعمال إحدى دوال المصفوفات (مثل `join()`) فلنسأل أنفسنا: هل يملك الكائن `myArray` المُنشَأ من الدالة البانية `Array()` نسخةً خاصةً به من الدالة `join()`؟ لنتحقق من ذلك (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = ['foo', 'bar'];

console.log(myArray.hasOwnProperty('join')); // الناتج: false

</script></body></html>
```

لا، لا يملك نسخةً خاصةً به من الدالة. لكن `myArray` يملك وصولًا إلى الدالة `join()` كما لو كانت خاصيةً تابعةً له. ما الذي يحدث هنا؟ حسنًا، لقد رأيتَ مثالًا حيًّا عن سلسلة prototype. إذ استطعنا الوصول إلى خاصيةٍ غيرُ موجودةٍ في الكائن `myArray` لكن تمكنت لغة JavaScript أن تعثر عليها في مكانٍ آخر. وهذا المكان مُحدَّدٌ جدًا؛ فلما أُنشِئَت الدالة البانية `Array()` في لغة JavaScript، أضيفت الدالة `join()` (بالإضافة لغيرها) كخاصيةٍ تابعة للخاصيةِ `prototype` للدالة `Array()`.

أكرِّر قولي أنَّك إذا حاولتَ الوصول إلى خاصيةٍ لا يملكها الكائن، فستبحث JavaScript في سلسلة prototype عن قيمةٍ لتلك الخاصية. وستنظر أولًا إلى الدالة البانية التي أنشأت الكائن (`Array` مثلًا)، ثم تنظر في الكائن الموجود في الخاصية `prototype` (مثلًا `Array.prototype`) لترى إن استطاعت أن تجد الخاصية هناك. إذا لم يحتوي أول كائن prototype على تلك الخاصية، فستستمر JavaScript ببحثها في السلسلة في الدالة البانية التي تقف خلف الدالة البانية البدائية؛ وستستمر في ذلك إلى أن تصل إلى نهاية السلسلة.

أين تنتهي السلسلة؟ لننظر إلى المثال مرةً أخرى، ونستدعي الدالة `toLocaleString()`

في myArray (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// Array.prototype و myArray لا تحتوي الكائنات
// toLocaleString() على الدالة
var myArray = ['foo', 'bar'];

// الدالة toLocaleString() مُعرّفة في
// Object.prototype.toLocaleString
console.log(myArray.toLocaleString()); // الناتج: 'foo,bar'

</script></body></html>
```

الدالة toLocaleString() غير معرفة ضمن الكائن myArray، لذا ستنظر JavaScript في سلسلة prototype وتبحث عن الخاصية في الكائن الموجود في الخاصية prototype للدالة البانية Array (أي: Array.prototype)، ولن تجدها هناك أيضًا، لذا ستبحث عن تلك الخاصية في خاصية prototype للدالة البانية Object() (أي Object.prototype)؛ وستجدها هناك؛ ولنفترض أنَّ الدالة غير موجودة في ذاك المكان، فستولِّد JavaScript خطأً يوضِّح أنَّ الخاصية غير معرَّفة (undefined).

ولمّا كانت جميع الخاصيات prototype عبارةٌ عن كائناتٍ، فإن آخر رابط في السلسلة هو Object.prototype. ولا توجد خاصية prototype تابعة لدالةٍ بانيةٍ أخرى يتوجب فحصها.

هنالك فصلٌ كاملٌ سنُقسِّم فيه سلسلة prototype إلى أجزاءٍ صغيرةٍ ليسهل فهمها، لذا إذا

وجدتَ نفسك ضائعًا تمامًا ولم تفهم شيئًا مما سبق، فاقرأ ذاك الفصل ثم ارجع إلى هنا لكي تُرسِّخ ما فهمته. أرجو أن تكون قد فهمت (من الشرح السابق المختصر) أنَّه عندما لا يُعثَر على خاصيةٍ ما (وتُعتَبَر undefined)، فستكون JavaScript قد بحثت في عدّة خاصيات prototype لتحديد أنَّ الخاصية غير معرفة (undefined). فعملية البحث السابقة تحدث دومًا، وهي تمثل طريقة تعامل لغة JavaScript مع الوراثة، وكيفية معرفتها لقيم الخاصيات.

6. استخدام الدالة hasOwnProperty للتحقق أنَّ خاصية أحد الكائنات تابعةٌ له

بينما يتحقق المعامل in من امتلاك العنصر لخاصيةٍ ما، بما في ذلك الخاصيات الموجودة في سلسلة prototype؛ فإنَّ الدالة hasOwnProperty تتحقق إن كان يملك الكائن خاصيةً ما أم يأخذها من سلسلة prototype.

سنتحقق في المثال الآتي أنَّ الكائن myObject يحتوي على الخاصية foo، ولا يرث الخاصية من سلسلة prototype؛ ولفعل ذلك سنسأل إذا امتلك الكائن myObject الخاصية foo (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject = {foo: 'value'};

// خاصية مملوكة للكائن
console.log(myObject.hasOwnProperty('foo')); // الناتج: true
```

```
// خاصية يرثها الكائن من سلسلة prototype
// الناتج: false
console.log(myObject.hasOwnProperty('toString'));

</script></body></html>
```

يجب استخدام الدالة hasOwnProperty إن احتجتَ إلى تحديد هل الخاصية تابعةٌ للكائن أم أنَّه يرثها من سلسلة prototype.

7. التحقق إن كان يحتوي الكائن على خاصية معيّنة باستخدام المعامل in

يُستخدَم المعامل in للتحقق (أي الناتج إما true وإما false) من أنَّ الكائن يحتوي على خاصيةٍ معيّنة. سنتحقق في المثال الآتي أنَّ foo خاصيةٌ موجودةٌ في الكائن myObject (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject = {foo: 'value'};
console.log('foo' in myObject); // الناتج: true

</script></body></html>
```

يجب أن تعلم أنَّ المعامل in لا يبحث في خاصيات الكائن المُحدَّد فحسب، وإنما في جميع

الخاصيات التي يرثها الكائن عبر سلسلة prototype. وبهذا ستبحث JavaScript بنفس الآلية التي شرحناها في القسم السابق، وإن لم تكن الخاصيةُ موجودةً في الكائن الحالي، فسيتم البحث عنها في سلسلة prototype.

وهذا يعني أنَّ الكائن `myObject` في المثال السابق يحتوي على الخاصية `toString` أيضًا وذلك عبر سلسلة prototype (أي `Object.prototype.toString`)، وحتى لو أنَّنا لم نُعرِّف تلك الخاصية للكائن مباشرةً (كقيمة أو كدالة. مثلًا `myObject.toString = 'foo'`)(مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject = {foo: 'value'};
console.log('toString' in myObject); // الناتج: true

</script></body></html>
```

نلاحظ في آخر شيفرة أنَّ الخاصية `toString` ليست تابعةً للكائن `myObject` مباشرةً؛ لكنه يرثها عبر `Object.prototype`، ولهذا السبب سيخبرنا المعامل `in` أنَّ الكائن `myObject` يملك الدالة `toString` الموروثة.

8. المرور على خاصيات الكائن باستخدام حلقة for in

باستخدام حلقة `for in`، سنتمكن من المرور على كل خاصية في كائنٍ ما. ففي الشيفرة الآتية سنستخدم الحلقة `for in` للحصول على أسماء الخاصيات الموجودة في الكائن `cody`

(مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var cody = {
    age : 23,
    gender : 'male'
};

// key هو متغير يُستخدَم لتمثيل اسم كل خاصية
for (var key in cody) {
    // لتفادي الخاصيات الموروثة من سلسلة prototype
    if(cody.hasOwnProperty(key)) {
        console.log(key);
    }
}

</script></body></html>
```

ملاحظات

- هنالك جانبٌ سلبيٌ لحلقة `for in`، حيث لن تمر على الخاصيات المملوكة للكائن فحسب، وإنما ستمر أيضًا على أيّة خاصيات يرثها الكائن (عبر سلسلة prototype)؛ وهذا يعني أنّك إن لم ترغب بهذه النتيجة (والأمر كذلك في أغلب الأحيان) فستحتاج إلى استخدام عبارة `if` الشرطية داخل الحلقة لكي تتحقق أنّك ستصل إلى الخاصيات التابعة للكائن الذي نريد الحصول على خاصياته فقط. ويمكن أن نفعل ذلك باستخدام الدالة `hasOwnProperty()`

التي تملكها كل الكائنات.

- الترتيب الذي تصل (access) فيه إلى الكائنات ليس موافقًا دائمًا للترتيب الذي تُعرَّف فيه داخل الحلقة؛ وأيضًا الترتيب الذي عرَّفتَ فيه الخاصيات ليس ضروريًا أن يوافق الترتيب الذي تمر فيه إليها.

- لا يمكن المرور إلا على الخاصيات في حلقة `for in`. على سبيل المثال، لن تظهر خاصية الدالة البانية. من الممكن أن تعرف ما هي الخاصيات التي يمكن المرور عليها باستخدام الدالة `propertyIsEnumerable()`.

9. كائنات المضيف والكائنات المُضمنة

أن يجب تعلم أنَّ البيئة (متصفح الويب على سبيل المثال) التي تُنفَّذ فيها شيفرات JavaScript تحتوي عادةً على ما يُعرَف بكائنات المضيف (host objects). لا تمثِّل كائنات المضيف جزءًا من تطبيق معيار ECMAScript، لكنها متوافرة ككائنات أثناء التنفيذ. يعتمد سلوك وإتاحة الوصول إلى كائنات المضيف كليًّا على البيئة التي يوفرها المضيف.

على سبيل المثال، في بيئة متصفح الويب يُعتبَر كائن `window` وجميع الكائنات التي يحتويها (ما عدا الكائنات المتوفرة من أساس لغة JavaScript) من كائنات المضيف.

سأريك خاصيات الكائن `window` في المثال الآتي (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// window إظهار جميع خاصيات الكائن
```

```
for (x in window) {
    console.log(x);
}

</script></body></html>
```

ربما تلاحظ أنَّ كائنات JavaScript المُضمنة فيها لا تُعتبَر ضمن كائنات المضيف، لأنَّ من الشائع أن تُفرِّق المتصفحات بين كائنات المضيف والكائنات المُضمنة في اللغة.

ولأنَّ تلك الكائنات خاصةٌ بمتصفحات الويب، فلا عجبَ أنَّ أحد أشهر تلك الكائنات يُمثِّل واجهةً للتعامل مع مستندات HTML (المعروفة أيضًا بشجرة DOM). سنُظهِر في المثال الآتي جميع الكائنات الموجودة ضمن الكائن `window.document` الذي توفره بيئة المتصفح (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// window.document إظهار جميع خاصيات الكائن
for (x in window.document) {
    console.log(x);
}

</script></body></html>
```

الذي أريد أنَّ أوضحه لك هنا هو أنَّ مواصفة JavaScript لا تهتم بكائنات المضيف (والعكس بالعكس). وهنالك خطٌ فاصلٌ بين ما توفره JavaScript (مثلًا إصدار 1.5 JavaScript

v3 ECMA-262 وإصـــــدارات Mozilla من JavaScript ذوات الأرقـــــام 1.6 و 1.7 و 1.8 و 1.8.1 و 1.8.5) وبين البيئة التي يوفرها المضيف، ولا يجوز الخلط بينهما.

ملاحظات	- توفِّر بيئة المضيف (مثلًا متصفح ويب) التي تُشغِّل شيفرات JavaScript عادةً كائنًا رئيسيًا (head object. مثلًا الكائن window في متصفح الويب) الذي تُخزَّن فيه أجزاءً من اللغة نفسها بالإضافة إلى كائنات المضيف (مثلًا window.location في متصفح الويب) إضافةً إلى الكائنات التي يُعرِّفها المستخدم (مثلًا الشيفرة التي تكتبها لتشغِّلها في متصفحك). - من الشائع أن تستعمل الشركة الصانعة لأحد متصفحات الويب والتي تستضيف مُفسِّر JavaScript نسخةً أحدث من JavaScript أو أن تُضيف ميزاتٍ من مواصفاتٍ مستقبليةٍ إلى JavaScript حتى لو لم يتم المصادقة عليها بعد (مثلًا إصدارات Mozilla من JavaScript ذوات الأرقام 1.6 و 1.7 و 1.8 و 1.8.1 و 1.8.5).

10. تحسين آلية التعامل مع الكائنات باستخدام مكتبة Underscore.js

إصـدار JavaScript 1.5 يفتقـر إلى بعض الوظـائف عنـدما يـأتي الـوقت للتعامـل مـع الكائنـات وإدارتها بكفاءة. إذا كنتَ تُشغِّل JavaScript على متصفح ويب، فأود أن أقترح عليك استخدام مكتبة Underscore.js عندما تحتاج إلى وظائف أكثر من تلك التي توفرها JavaScript 1.5. تعطيك مكتبة Underscore.js الوظائف الآتية عندما تتعامل مع الكائنات.

هذه الدوال تعمل على كل الكائنات والمصفوفات:

- `each()`
- `map()`
- `reduce()`
- `reduceRight()`
- `detect()`
- `select()`
- `reject()`
- `all()`
- `any()`
- `include()`
- `invoke()`
- `pluck()`
- `max()`
- `min()`
- `sortBy()`
- `sortIndex()`
- `toArray()`
- `size()`

وهذه الدوال تعمل على كل الكائنات:

- `keys()`
- `values()`
- `functions()`

- `extend()`
- `clone()`
- `tap()`
- `isEqual()`
- `isEmpty()`
- `isElement()`
- `isArray()`
- `isArguments()`
- `isFunction()`
- `isString()`
- `isNumber()`
- `isBoolean()`
- `isDate()`
- `isRegExp()`
- `isNaN()`
- `isNull()`
- `isUndefined()`

تعجبني هذه المكتبة لأنّها تستفيد من ميزات JavaScript الجديدة التي تدعمها المتصفحات الحديثة، ولكنها بنفس الوقت توفر نفس الوظيفة إلى المتصفحات التي لا تحتوي تلك التحديثات، وكل ذلك دون تغيير البنية الأساسية للغة JavaScript إلا إن اقتضت الضرورة.

ملاحظة	قبل أن تبدأ باستخدام Underscore.js، تأكَّد أنَّ الوظائف التي تريدها لا توفرها أيّة مكتبة JavaScript (أو إطار عمل) تستخدمها في شيفراتك (مثل jQuery).

الفصل الثالث:

الكائن ()Object

3

1. لمحة نظرية عن استخدام كائنات ()Object

باستعمالنا للدالة البانية المُضمَّنة في اللغة `Object()`، نتمكن من إنشاء كائنات عمومية (generic) فارغة. وفي الواقع إذا كنتَ تذكر بداية الفصل الأول، فهذا ما فعلناه عندما أنشأنا الكائن cody. لنُنشِئ الكائن cody مرةً أخرى (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// إنشاء كائن فارغ دون أيّة خاصيات
var cody = new Object();

// للتأكّد أنّ الكائن cody هو كائن عمومي فارغ
for (key in cody) {
    if(cody.hasOwnProperty(key)) {
        // لا يجب أن تشاهد أي ناتج لعدم وجود خاصيات
        console.log(key);
    }
}

</script></body></html>
```

كل ما فعلناه هنا هو استخدام الدالة البانية `Object()` لإنشاء كائن عمومي اسمه `cody`.

يمكن أن تتخيّل أنّ الدالة البانية `Object()` هي قالبُ تقسيمٍ للكعكات لإنشاء كائنات فارغة لا تملك أيّة خاصيات أو دوال مُعرفّة مسبقًا (طبعًا ما عدا تلك التي ترثها من سلسلة prototype).

ملاحظة	إن لم يكن ذلك واضحًا لك، فأود أن أنوّه أنَّ الدالة البانية ‎`Object()`‎ هي كائنٌ بحد ذاتها. أي أنَّ الدالة البانية مبينةٌ على كائنٍ أُنشِئَ من الدالة البانية ‎`Function`‎؛ وقد يبدو الأمر مُربِكًا. لكن تذكَّر أنَّ الأمر مشابهٌ للدالة البانية ‎`Array`‎، لكن الدالة البانية ‎`Object`‎ تُنشِئ كائناتٍ فارغة. ولا تغفل عن إمكانية إنشائك لأي عدد من الكائنات الفارغة، لكن إنشاء كائن فارغ مثل ‎`cody`‎ يختلف كثيرًا عن إنشاء دالة بانية خاصة بك مع وضع خاصيات مُعرَّفة مسبقًا معها. عليك أن تفهم أنَّ الكائن ‎`cody`‎ هو مجرد كائن بسيط أنشأناه من الدالة البانية ‎`Object()`‎؛ وللاستفادة من قدرات لغة JavaScript، فعليك ألّا تستوعب طريقة إنشاء كائنات فارغة من الدالة البانية ‎`Object()`‎ فحسب، وإنما كيفية إنشاء «صنف» خاص من الكائنات (مثلًا ‎`Person()`‎) عبر دالة بانية شبيهة بالدالة البانية ‎`Object()`‎.

2. معاملات الدالة البانية ()Object

تأخذ الدالة البانية ‎`Object()`‎ معاملًا (parameter) اختياريًا وحيدًا؛ وهذا المعامل هو القيمة التي تريد إنشاءها، وإن لم تُحدِّدها فسيُعتبَر أنَّ قيمتها هي ‎`null`‎ أو ‎`undefined`‎ (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// إنشاء كائن فارغ دون أيّة خاصيات
var cody1 = new Object();
var cody2 = new Object(undefined);
var cody3 = new Object(null);
```

```
// الناتج: 'object object object'
console.log(typeof cody1, typeof cody2, typeof cody3);

</script></body></html>
```

إذا مُرِّرَت قيمةٌ عدا null أو undefined إلى الدالة البانية ()Object، فستُنشَأ القيمة المُمرَّرة ككائن. لذا نظريًا يمكننا استخدام الدالة البانية ()Object لإنشاء أيّة كائنات مُضمَّنة في اللغة التي تملك دوالًا بانية. انظر المثال الآتي للتوضيح (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

/*
 استخدام الدالة البانية Object() لإنشاء سلسلة نصية وعدد
ومصفوفة ودالة وقيمة منطقية وكائن تعابير نمطية.
*/

// الناتج يؤكد عملية الإنشاء
console.log(new Object('foo'));
console.log(new Object(1));
console.log(new Object([]));
console.log(new Object(function() {}));
console.log(new Object(true));
console.log(new Object(/\bt[a-z]+\b/));

/*
```

```
من غير الشائع إنشاء سلسلة نصية وعدد ومصفوفة ودالة وقيمة
منطقية وكائن تعابير نمطية عبر الدالة البانية ()Object؛
لكنني أريد أن أبيّن لك إمكانية فعل ذلك.
*/

</script></body></html>
```

3. الخاصيات والدوال الموجودة في ()Object

يملك الكائن `Object()` الخاصيات الآتية (باستثناء الخاصيات والدوال التي يرثها):

- الخاصيات (مثلًا `Object.prototype`):
  - `prototype`

4. الخاصيات والدوال الموجودة في الكائنات من نوع ()Object

تملك الكائنات ذات النوع `Object()` الخاصيات والدوال الآتية (باستثناء الخاصيات والدوال التي ترثها):

- الخاصيات (مثلًا `var myObject = {}; myObject.constructor;`):
  - `constructor`
- الدوال (مثلًا `var myObject = {}; myObject.toString();`):
  - `hasOwnProperty()`
  - `isPrototypeOf()`
  - `propertyIsEnumerable()`

- `toLocaleString()`
- `toString()`
- `valueOf()`

ملاحظة

تنتهي سلسلة prototype في `Object.prototype` وهذا يعني أنَّ جميع الخاصيات والدوال التابعة للكائن `Object()` (الظاهرة أعلاه) ستورَّث إلى جميع كائنات JavaScript.

5. إنشاء كائنات ()Object بالطريقة المختصرة

إنشاء كائنات `Object()` بالطريقة المختصرة يعني إنشاء كائن باستخدام الأقواس المعقوفة مع أو بدون خاصيات (مثلًا `var cody = {};`). هل تتذكر في بداية الفصل الأول عندما أنشأنا الكائن cody ثم أعطيناه الخصائص باستخدام طريقة النقط؟ لنفعل ذلك مرةً أخرى (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var cody = new Object();
cody.living = true;
cody.age = 33;
cody.gender = 'male';
cody.getGender = function() {return cody.gender;};

console.log(cody); // ستظهر الخصائص والدوال التابعة للكائن
```

```
</script></body></html>
```

لاحظ أنَّ إنشاء الكائن cody وإعطاؤه قيمًا لخاصياته أخذ منا خمسة تعبيرات برمجية؛ أمّا باستخدام الطريقة المختصرة فسنتمكن من إنشاء كائن cody السابق بتعبيرٍ برمجيٍ وحيد (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var cody = {
    living: true,
    age: 23,
    gender: 'male',
    getGender: function() {return cody.gender;}
};
// لاحظ عدم وجود فاصلة بعد آخر خاصية

console.log(cody); // ستظهر الخصائص والدوال التابعة للكائن

</script></body></html>
```

سيمكننا استخدام الشكل المختصر من إنشاء الكائنات وتعريف خاصياتها وإسناد قيم إليها باستخدام شيفرات أقل مع إظهار البيانات بطريقة تجعل قراءتها أسهل. لاحظ استخدام المعاملَين : و , في تعبيرٍ وحيد. يجدر بالذكر أنَّ هذه هي الصيغة المستحسنة لإنشاء الكائنات في JavaScript لأنَّها أقصر وقراءتها أسهل.

يجب أن تعلم أيضًا أنَّه بالإمكان التعبير عن أسماء الخاصيات كسلاسل نصية (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var cody = {
    'living': true,
    'age': 23,
    'gender': 'male',
    'getGender': function() {return cody.gender;}
};

console.log(cody); // ستظهر الخصائص والدوال التابعة للكائن

</script></body></html>
```

ليس من الضروري تعريف أسماء الخاصيات كسلاسل نصية ما لم يكن اسم الخاصية:

- أحد الكلمات المحجوزة (مثلًا class)
- يحتوي على فراغات أو على محارف خاصة (أي شيء ما عدا الأرقام والأحرف وإشارة الدولار [$] أو الشرطة السفلية [_])
- يبدأ برقم

تحذير	احذر من وجود فاصلة بعد آخر خاصية من خاصيات الكائن، لأنَّ هذا قد يُسبِّب خطأً في بعض بيئات JavaScript.

6. جميع الكائنات ترث من Object.prototype

الدالة البانية `Object()` في JavaScript لها خصوصيتها، لأنَّ خاصية `prototype` التابعة لها هي آخر محطة في سلسلة prototype ‏(prototype chain).

سأغيّر في المثال الآتي الكائن `Object.prototype` لأضيف الخاصية `foo`، ثم سأنشِئ سلسلةً نصيةً وسأحاول الوصول إلى الخاصية `foo` كما لو أنها خاصية تابعة للسلسلة النصية. ولمّا كان الكائن `myString` لا يملك الخاصية `foo`، فستبحث JavaScript عن قيمة تلك الخاصية في `String.prototype` وإن لم تجدها (ولن تجدها)، فستنظر في المحطة التالية ألا وهي `Object.prototype` (التي هي آخر محطة تبحث فيها JavaScript عن قيمةٍ ما). وسنعثر على القيمة `foo` هناك لأنني أضفتها، وبالتالي ستُعاد قيمة الخاصية `foo` (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

Object.prototype.foo = 'foo';

var myString = 'bar';

// الناتج هو 'foo' الموجود في Object.prototype.foo
// والذي حصلنا على قيمته عبر سلسلة prototype
console.log(myString.foo);

</script></body></html>
```

تحذير

انتبه! إضافة أي شيء إلى `Object.prototype` سيؤدي إلى ظهوره في حلقة `for in` وفي سلسلة prototype، ولهذا السبب يُقال أنَّ تغيير الكائن `Object.prototype` هو أمرٌ غير مستحسن ولا ينصح بفعله.

الفصل الرابع:

الكائن ()Function

4

1. لمحة نظرية عن استخدام كائنات ()Function

الدالة هي حاوية لتعليمات برمجية التي تُستدعى باستخدام المعامل `()`. يمكن أن تُمرِّر المعاملات (parameters) إلى داخل القوسين أثناء عملية الاستدعاء لكي تتمكن التعليمات البرمجية الموجودة داخل الدالة من الوصول إلى قيمٍ معيّنة عندما تعمل الدالة.

سننشِئ في المثال الآتي نسختين من الدالة `addNumbers`، واحدة تستعمل المعامل `new` والأخرى تستعمل الشكل المبسط الأكثر شهرةً. وتتوقع كلا الدالتين استقبال معاملَين؛ وسنستدعي الدالتين وسنمرر الوسيطين عبر المعامل `()` (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var addNumbersA = new Function('num1', 'num2', 'return num1 +
num2');

console.log(addNumbersA(2, 2)); // الناتج: 4

// ويمكن أيضًا أن تُكتَب الدالة بالشكل المختصر، وهو أشهر
var addNumbersB = function(num1, num2) {return num1 + num2;};

console.log(addNumbersB(2, 2)); // الناتج: 4

</script></body></html>
```

يمكن أن تُعيد الدالة قيمةً أو تبني كائنًا أو يمكن أن نستعملها كآليةٍ تنظيميةٍ لتنفيذ الشيفرات

البرمجية فقط. هنالك استعمالاتٌ عديدة للدوال في JavaScript، لكن في أبسط الأشكال نقول: الدالةُ هي مجالٌ فريدٌ (unique scope) من التعليمات البرمجية القابلة للتنفيذ.

2. معاملات الدالة البانية ()Function

يمكن للدالة البانية `Function()` أن تأخذ عددًا لا حصر له من المعاملات، لكن آخر معامل تتوقع الدالة البانية `Function()` استقباله هو سلسلةٌ نصيةٌ تحتوي على التعليمات البرمجية التي تُشكِّل وتألّف جسم الدالة. أيّة مُعاملات أخرى تُمرِّر إلى الدالة البانية قبل آخر معامل ستُتاح إلى الدالة التي ستُنشَأ كمعاملات. ومن الممكن أيضًا إرسال عدّة معاملات باستخدام سلسلة نصية تُفصَل القيم فيها بفاصلة.

سأبيّن في المثال الآتي استخدام الدالة البانية `Function()` مع عدّة أنماط لإنشاء كائن دالة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var addFunction = new Function('num1', 'num2', 'return num1 +
num2');

/*
 بشكلٍ بديل، يمكن أن تُمرّر سلسلة نصية تُفصَل القيم فيها بفاصلة
كأول معامل للدالة البانية، ويليها جسم الدالة.
*/
var timesFunction = new Function('num1,num2', 'return num1 *
num2');
```

```
// الناتج: '2 2'
console.log(addFunction(2,2),timesFunction(2,2));

// ويمكن أيضًا أن تُكتَب الدالة بالأشكال الأشهر لتعريف الدوال


// تعريف دالة بإسنادها إلى متغير
var addFunction = function(num1, num2) {return num1 + num2;};

// تعريف دالة باستخدام تعبير برمجي مخصص لهذا الغرض
function addFunction(num1, num2) {return num1 + num2;}

</script></body></html>
```

ملاحظات

- ليس من المستحسن استدعاء الدالة البانية `Function()` مباشرةً، ولا أحد يفعل ذلك إطلاقًا لأنَّ JavaScript ستستعمل الدالة `eval()` لتفسير السلسلة النصية التي تحتوي على البنية المنطقية للدالة؛ ويعتبر الكثيرون أنَّ استخدام `eval()` هو تعقيدٌ زائدٌ عن اللزوم، وإن استخدمِتَ، فمكن المرجع حدوث مشكلة في البنية التصميمية للشيفرة البرمجية.

- استخدام الدالة البانية `Function()` دون الكلمة المحجوزة new سيعطي نفس تأثير استخدام الدالة البانية لإنشاء كائنات للدوال (مثلًا `new Function('x', 'return x')` أو `Function('x', 'return x')`).

- لن يُنشَأ «تعبيرٌ مغلق» (closure) (انظر الفصل السابع) عند استدعاء الدالة البانية `Function()` مباشرةً.

3. الخاصيات والدوال الموجودة في ()Function

يملك الكائن `Function()` الخاصيات الآتية (باستثناء الخاصيات والدوال التي يرثها):

- الخاصيات (مثلًا `Function.prototype`):
  - `prototype`

4. الخاصيات والدوال الموجودة في الكائنات من نوع ()Function

تملك الكائنات ذات النوع `Function()` الخاصيات والدوال الآتية (باستثناء الخاصيات والدوال التي ترثها):

- الخاصيات (مثلًا
```
var myFunction = function (x, y, z) {};
myFunction.length;
```
):
  - `arguments`
  - `constructor`
  - `length`
- الدوال (مثلًا
```
var myFunction = function (x, y, z) {};
myFunction.toString();
```
):
  - `apply()`

- `call()`
- `toString()`

5. تُعيد الدوال دومًا قيمةً ما

صحيحٌ أنَّه بالإمكان إنشاء دالة بسيطة التي تُنفِّذ التعليمات الموجودة فيها، لكن من الشائع أن تُعيد الدالة قيمةً. ستُعيد في المثال الآتي سلسلة نصيةً من الدالة `sayHi` (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var sayHi = function() {
    return 'Hi';
};

console.log(sayHi()); // الناتج: "Hi"

</script></body></html>
```

إذا لم يتم تحديد قيمة لكي تُعيدها الدالة، فستعيد القيمة `undefined`. سأستدعي في المثال الآتي الدالة `yelp` التي تكتب السلسلة النصية `'yelp'` دون الحاجة إلى تحديد قيمة لكي تُعيدها (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var yelp = function() {
```

```
    console.log('I am yelping!');
   // ستعيد الدوال القيمة undefined إن لم نُحدِّد قيمةً لتعيدها
}

// الناتج true
// لأنّه يجب دومًا إعادة قيمة، حتى لو لم نُحدِّد واحدةً
console.log(yelp() === undefined);

</script></body></html>
```

الفكرة هنا هي أنَّ كل دالة يجب أن تعيد قيمةً، حتى لو لم نُصرِّح أو نوفِّر قيمةً معيّنةً؛ فإن لم تُحدِّد قيمةٌ لتعيدها الدالة، فستُعاد القيمة `undefined`.

6. ليست الدوال إحدى البنى البرمجية فحسب وإنما تُمثِّل قيمًا

الدوال في JavaScript عبارةٌ عن كائنات؛ وهذا يعني أنَّ أيّة دالة يمكن أن تُخزَّن في متغير أو في مصفوفة أو في كائن؛ ويمكن أيضًا تمريرها أو إعادتها من دالةٍ أخرى. وأيضًا تملك الدالة خاصياتٍ (properties) لأنها كائنات (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// هذه دوال مُخزّنة في متغيرات (funcA)
// ومصفوفات (funcB) وكائنات (funcC)

// ستستدعى كالتالي: funcA()
```

```
var funcA = function(){};
// funcB[0]() :ستستدعى كالتالي
var funcB = [function(){}];
// funcC['method']() أو funcC.method()
var funcC = {method: function(){}};

// يمكن أن تُمرّر الدوال وأن تُعاد من دوالٍ أخرى
var funcD = function(func){
    return func;
};

var runFuncPassedToFuncD = funcD(function()
{console.log('Hi');});

runFuncPassedToFuncD();

// تذكّر أنّ الدوال هي كائنات
// وهذا يعني أنّها تستطيع امتلاك خاصيات
var funcE = function(){};
funcE.answer = 'yup'; // خاصية تابعة لنسخة الكائن
console.log(funcE.answer); // الناتج: 'yup'

</script></body></html>
```

من المهم جدًا أن تستوعب أنَّ الدالة هي كائن، وبالتالي هي قيمة؛ ويمكن تمريرها أو تعديلها كما في باقي التعبيرات البرمجية في JavaScript.

7. تمرير المعاملات إلى دالة

يمكننا أن نُشبِّه المعاملات (parameters) بالشاحنات التي تنقل القيم إلى مجال (scope) الدالة عندما تُستدعى. استدعينا في المثال الآتي الدالة `addFunction()`، ولأنَّا عرَّفناها مسبقًا مع معاملَين، فستصبح القيمتان المُمرَّرتان إليها متاحتَين ضمن مجالها (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var addFunction = function(number1, number2) {
    var sum = number1 + number2;
    return sum;
}
console.log(addFunction(3, 3)); // الناتج: 6

</script></body></html>
```

ملاحظات	- على النقيض من بعض لغات البرمجة الأخرى: من السليم تمامًا في JavaScript عدم تحديد قيمة للمعاملات حتى لو عُرِّفَت الدالة لكي تستقبل معاملات. ببساطة ستُعطى القيمة `undefined` لأي معامل غير مُحدَّد. وبكل تأكيد، إن لم تُحدِّد قيمًا للمعاملات، فقد لا تعمل الدالة بشكل صحيح. - إذا مررتَ عددًا غير متوقعٍ من المعاملات لدالة (التي لم تُعرِّفها عند إنشائك للدالة)، فلن يحدث أي خطأ؛ ومن الممكن أن تصل إلى تلك المعاملات عبر الكائن `arguments`، الذي هو متاحٌ لجميع الدوال.

8. القيمتان this و arguments متاحتان لجميع الدوال

تتاح القيمتان this و arguments في جسم/مجال جميع الدوال.

الكائن arguments هو كائنٌ شبيه بالمصفوفات يحتوي على كل المعاملات التي مُرِّرَت إلى الدالة. فحتى لو نسينا -في المثال التالي- تحديد معاملات للدالة عند تعريفها، فسنتمكن من الاعتماد على المصفوفة arguments الموجودة في الدالة للوصول إلى المعاملات التي مُرِّرَت إليها أثناء استدعائها (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var add = function() {
    return arguments[0] + arguments[1];
};

console.log(add(4, 4)); // الناتج: 8

</script></body></html>
```

أما الكلمة المحجوزة this (الموجودة في كل الدوال) فهي مرجعية إلى الكائن الذي يحتوي الدالة. وكما قد تتوقع، الدوال الموجودة ضمن كائنات كخاصيات يمكن أن تَستعمِل this للحصول على مرجع إلى الكائن «الأب»؛ أما عندما تُعرَّف الدالة في المجال العام (global scope)، فستكون قيمة this هي الكائن العام. راجع الشيفرة الآتية لإزالة الغموض عن القيمة التي تُعيدها الكلمة المحجوزة this في مختلف الحالات (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject1 = {
    name: 'myObject1',
    myMethod: function(){console.log(this);}
};

myObject1.myMethod(); // الناتج: 'myObject1'

var myObject2 = function(){console.log(this);};

myObject2(); // الناتج: Window

</script></body></html>
```

9. الخاصية arguments.callee

يملك الكائن arguments خاصيةً باسم callee التي تُشير إلى الدالة التي تُنفَّذ حاليًا. يمكن استخدام هذه الخاصية للإشارة إلى الدالة ضمن مجال تنفيذ (scope) الدالة (مثلًا arguments.callee) أي ستشير إلى الدالة نفسها. سنحصل في المثال الآتي على مرجع للدالة التي تُنفَّذ (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = function foo() {
```

```
    console.log(arguments.callee); // الناتج: foo()
    // يمكن استخدام الخاصية callee لاستدعاء
    // الدالة foo تعاوديًا (recursively)
    // arguments.callee()
}();

</script></body></html>
```

يمكن الاستفادة من هذه الخاصية عندما نحتاج إلى استدعاء الدالة تعاوديًا (recursively).

10. الخاصية length والخاصية arguments.length

يملك الكائن `arguments` خاصيةً فريدةً هي الخاصية `length`؛ وربما تظن أنَّ هذه الخاصية ستعطيك عدد المعاملات المُعرَّفة في الدالة، لكنها ستعطيك في الواقع عدد الوسائط المُمرَّرة إلى الدالة التي أُستدعيت (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myFunction = function(z, s, d) {
    return arguments.length;
};

// الناتج 0 لأنّه لم يُمرّر أيّ وسيط إلى الدالة
console.log(myFunction());
```

```
</script></body></html>
```

استخدام الخاصية length التابعة لجميع النسخ من الكائن ()Function سيعطينا العدد الكلي للمعاملات التي تتوقع الدالة استقبالها (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myFunction = function(z, s, d, e, r, m, q) {
    return myFunction.length;
};

console.log(myFunction()); // الناتج: 7

</script></body></html>
```

ملاحظة

أصبحت الخاصية arguments.length مهملةً (deprecated) بدءًا من الإصدار JavaScript 1.4، ولكن ما زال بالإمكان الوصول إلى عدد الوسائط المُمرَّر إلى الدالة عبر الخاصية length لكائن الدالة. لذا في المستقبل عليك الحصول على عدد الوسائط عبر استعمال الخاصية callee بعد الحصول على مرجع للدالة التي أُستدعيت (أي arguments.callee.length).

11. إعادة تعريف معاملات الدالة

يمكن إعادة تعريف معاملات الدالة داخل الدالة مباشرةً أو باستخدام المصفوفة

arguments. ألقِ نظرةً على هذه الشيفرة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = false;
var bar = false;

var myFunction = function(foo, bar) {
    arguments[0] = true;
    bar = true;
    console.log(arguments[0], bar); // الناتج: true true
}

myFunction();

</script></body></html>
```

لاحظ كيف استطعت إعادة تعريف المعامل bar باستعمال ترتيبه (index) في مصفوفة arguments أو مباشرةً بإسناد قيمة جديدة إلى المعامل.

12. إعادة قيمة من الدالة قبل انتهاء تنفيذها (أي إلغاء تنفيذ الدالة)

يمكن إلغاء تنفيذ الدوال في أيّ وقتٍ أثناء تنفيذها باستخدام الكلمة المحجوزة return مع أو بدون قيمة. سألغي في الشيفرة الآتية تنفيذ الدالة add إذا لم تُحدَّد قيمة المعامل أو لم يكن

رقمًا (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var add = function(x, y) {
    // إعادة رسالة خطأ إن لم تكن المعاملات رقميةً
    if (typeof x !== 'number' || typeof y !== 'number')
    {return 'pass in numbers';}
    return x + y;
}
console.log(add(3,3)); // الناتج: 6
console.log(add('2','2')); // الناتج: 'pass in numbers'

</script></body></html>
```

الفكرة التي أريد إيصالها هي أنَّك تستطيع إلغاء تنفيذ الدالة باستخدام الكلمة المحجوزة return في أيّ مرحلة من مراحل تنفيذ الدالة.

13. تعريف الدالة (دالة بانية، أو عبر تعليمة برمجية، أو عبر تعبير برمجي)

يمكن تعريف الدالة بثلاث طرائق: عبر استخدام دالة بانية، أو عبر تعليمة برمجية (statement) أو عبر تعبير برمجي (expression). سأشرح الطرائق جميعها في المثال الآتي (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

/*
 عند استخدام الدالة البانية، سيكون آخر وسيط هو جسد الدالة
 الذي يحتوي على التعليمات البرمجية، وسيُعامَل كل شيء قبلها
 كمعامل.
*/
var addConstructor = new Function('x', 'y', 'return x + y');

// تعليمة برمجية
function addStatement(x, y) {
    return x + y;
}

// تعبير برمجي
var addExpression = function(x, y) {
    return x + y;
};

// الناتج: 4 4 4
console.log(addConstructor(2,2), addStatement (2,2),
addExpression (2,2));

</script></body></html>
```

ملاحظة

يقول البعض أنَّ هنالك طريقةٌ رابعةٌ لتعريف الدوال، وتسمى «التعبير البرمجي المُسمى» (named expression)، وهذه الطريقة تشبه طريقة تعريف الدالة عبر تعبير برمجي إلا أنها تتضمن أيضًا اسمًا للدالة (مثلًا `var add = function add(x, y) {return x+y}`).

14. استدعاء الدالة (كدالة عادية، أو كدالة في كائن، أو كدالة بانية، أو عبر ()call و ()apply)

يمكن استدعاء الدوال بأربع طرائق أو أنماط:

- كدالة عادية
- كدالة في كائن
- كدالة بانية
- باستخدام `apply()` أو `call()`

سأستعرض كلًا من طرائق الاستدعاء السابقة في المثال الآتي (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// استدعاؤها كدالة
var myFunction = function(){return 'foo'};
console.log(myFunction()); // الناتج: 'foo'
```

```
// استدعاؤها كدالة في كائن
var myObject = {myFunction: function(){return 'bar';}}
console.log(myObject.myFunction()); // الناتج: 'bar'

// استدعاؤها كدالة بانية
var Cody = function(){
    this.living = true;
    this.age = 33;
    this.gender = 'male';
    this.getGender = function() {return this.gender;};
}

// استدعاء الدالة البانية لإنشاء الكائن
var cody = new Cody();
console.log(cody); // إظهار خاصيات ودوال الكائن

// استخدام apply() و call()
var greet = {
    runGreet: function(){
        console.log(this.name,arguments[0],arguments[1]);
    }
}

var cody = {name:'cody'};
var lisa = {name:'lisa'};
```

```
// cody استدعاء الدالة runGreet كما لو أنها داخل الكائن
greet.runGreet.call(cody,'foo','bar'); // الناتج: cody

// lisa استدعاء الدالة runGreet كما لو أنها داخل الكائن
greet.runGreet.apply(lisa, ['foo','bar']); // الناتج: lisa

/*
لاحظ الفرق بين ()call و ()apply في كيفية إرسال المعاملات إلى
 الدالة التي تُستدعى
*/

</script></body></html>
```

احرص على أن تتعرّف على جميع الأنواع الأربعة من طرائق الاستدعاء، لأنَّ الشيفرات التي قد تتعامل معها في المستقبل يمكن أن تحتوي على أيٍّ منها.

15. الدوال المجهولة

الدالة المجهولة (anonymous function) هي الدالة التي لم يُعطى لها مُعرِّف (identifier).

تُستخدَم الدوال المجهولة في غالبية الأحيان لتمرير دالة كمعامل إلى دالةٍ أخرى (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// دالة مجهولة، لكن لا توجد طريقة لاستدعائها
// function(){console.log('hi');};
```

```
// إنشاء دالة تستطيع استدعاء دالة مجهولة
var sayHi = function(f){
    f(); // استدعاء الدالة المجهولة
}

// تمرير دالة مجهولة كمعامل
sayHi(function(){console.log('hi');}); // الناتج: 'hi'

</script></body></html>
```

16. الدوال المُعرَّفة في تعبير برمجي التي تستدعي نفسها مباشرةً

يمكن للدوال المعرَّفة في تعبيرٍ برمجي (أي أيّة دالة ما عدا تلك المُنشَأة من الدالة البانية `Function()`) أن تُستدعى مباشرةً بعد تعريفها باستخدام معامل الأقواس `()`. سنُنشِئ في المثال الآتي الدالة `sayWord()` المُعرَّفة في تعبيرٍ برمجي ثم سنستدعيها مباشرةً. وهذا ما ندعوه «بالدوال التي تستدعي نفسها» (self-invoking function) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// الناتج: 'Word 2 yo mo!'
var sayWord = function() {console.log('Word 2 yo mo!');}();
```

```
</script></body></html>
```

17. الدوال المجهولة التي تستدعي نفسها مباشرةً

من الممكن إنشاء عبارة برمجية تُمثِّل دالةً مجهولةً تستدعي نفسها مباشرةً. وهي تسمى «الدوال المجهولة التي تستدعي نفسها» (self-invoking anonymous function). سأريك في المثال الآتي عدّة أشكال[3] لدوال مجهولة تستدعي نفسها مباشرةً (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// أكثر الأشكال استخدامًا بين المبرمجين
(function(msg) {
    console.log(msg);
})('Hi');

// هذا الشكل مختلف قليلًا، لكن يؤدي نفس الغرض
(function(msg) {
    console.log(msg)
}('Hi'));

// أكثر شكل مختصر
!function sayHi(msg) {console.log(msg);}('Hi');

// لمعلوماتك: لا يمكن استدعاء الدالة المجهولة بهذا الشكل
```

3 راجع صفحة «Immediately-invoked function expression» في ويكيبيديا لمزيدٍ من المعلومات.

```
// function sayHi() {console.log('hi');}();

</script></body></html>
```

ملاحظة

وفقًا لمعيار ECMAScript، الأقواس حول الدالة (أو أي شيء يحوّل الدالة إلى تعبير [expression]) مطلوبةٌ إذا كانت ستُستدعى الدالة مباشرةً.

18. يمكن تشعّب الدوال

يمكن تشعّب الدوال وتعريفها داخل بعضها دون أي حدود. سأغلّف في المثال الآتي الدالة goo داخل الدالة bar داخل الدالة foo (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = function() {
    var bar = function() {
        var goo = function() {
            // window إظهار مرجعية إلى الكائن الرئيسي
            console.log(this);
        }();
    }();
}();

</script></body></html>
```

الفكرة التي أريد إيصالها هنا بسيطة، وهي أنَّه يمكن تشعب الدوال وتعريفها داخل بعضها دون وجود قيود على «عمق» التشعب.

ملاحظة	تذكر أنَّ قيمة this في الدوال المتشعبة هو الكائن الرئيسي (مثلًا الكائن window في متصفحات الويب) في إصدار JavaScript 1.5 EMCA-262 v3.

19. تمرير الدوال إلى الدوال وإعادة الدوال من الدوال

كما ذكرنا سابقًا أنَّ الدالة في JavaScript تُمثِّل قيمةً، ويمكن أن يُمرَّر إلى الدالة أيُّ نوعٍ من المعاملات، وبالتالي يمكن تمرير دالة إلى دالة أخرى. والدوال التي تقبل دوالًا أخرى كوسائط أو تُعيدها تسمى أحيانًا «higher-order functions».

سأُمرِّر في المثال الآتي دالةً مجهولةً إلى الدالة foo والتي سوف نُعيدها مباشرةً من الدالة foo. وبهذا سيُشير المتغير bar إلى الدالة المجهولة لأنَّ الدالة foo تستقبل الدالة المجهولة وتعيدها مباشرةً (تمعّن في الشيفرة لفهم ما سبق فهمًا كاملًا) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// يمكن أن تُمرّر الدوال أو تُعاد من الدوال الأخرى
var foo = function(f) {
    return f;
}

var bar = foo(function() {console.log('Hi');});
```

```
bar(); // الناتج: 'Hi'

</script></body></html>
```

عندما تُستدعى الدالة bar، فستستدعي الدالة المجهولة التي مررناها إلى الدالة ()foo ومن ثم ستُعاد الدالة المجهولة من الدالة ()foo وستُسنَد إلى المتغير bar. وهذا المثال يوضِّح كيف يمكن تمرير الدوال كغيرها من القيم.

20. استدعاء الدوال قبل تعريفها

يمكن لدالة مُعرَّفة عبر تعبير برمجي أنَّ تُستدعى أثناء التنفيذ قبل تعريفها. ربما ترى أنَّ الأمر غريبٌ بعض الشيء، لكن يجب أن تعلم ذلك لكي تستفيد منه، أو على الأقل أن تفهم ما الذي يجري عندما تواجه شيفرةً تفعل ذلك. استدعيتُ في المثال الآتي الدالة ()sayYo و ()sum قبل تعريفها (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// المثال الأول

var speak = function() {
    // لم تُعرّف الدالة sayYo() بعد، لكن يمكن استدعاؤها
    // الناتج: yo
    sayYo();
```

```
    function sayYo() {console.log('Yo');}
}(); // استدعاء الدالة مباشرةً

// المثال الثاني

// لم تُعرّف الدالة sum() بعد، لكن يمكن استدعاؤها
console.log(sum(2, 2)); // الناتج: 4
function sum(x, y) {return x + y;}

</script></body></html>
```

تُفسَّر تعليمات إنشاء الدوال وتُضاف إلى «مكدس التنفيذ»[4] (execution stack) قبل أن تُنفَّذ الشيفرة. عليك أن تضع ذلك بعين الاعتبار عند استخدامك لتعليمات إنشاء الدوال.

ملاحظة	الدوال المُعرَّفة في تعبيرات برمجية (function expressions) لا يمكن استدعاؤها قبل تعريفها، يمكن استدعاء الدوال المُعرَّفة بتعليمات برمجية (function statements) قبل تعريفها فقط.

21. يمكن للدالة أن تستدعي نفسها (التنفيذ التعاودي)

من المسموح في لغة JavaScript أن تستدعي الدالة نفسها. وفي الحقيقة أنَّ هذه النمط من أنماط البرمجة الشائعة. ستُنشِئ في المثال الآتي الدالة `countDownForm`، التي تستدعي نفسها

4 يُسمى أيضًا «مكدس الاستدعاء» (call stack)، وهو مكدس يُخزِّن معلومات عن الدوال قيد التنفيذ. راجع صفحة ويكيبيديا «Call stack» لمزيدٍ من المعلومات.

عبر اسم الدالة (countDownForm). بشكل أساسي، تُنشِئ هذه الدالة حلقة تكرار تعد من 5 إلى 0 (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var countDownFrom = function countDownFrom(num) {
    console.log(num);
    num--; // تغيير قيمة المعامل
    // إذا كان num < 0 فسيتم إنهاء الدالة مباشرةً
    if (num < 0){return false;}

    // كان بإمكاننا أيضًا استدعاء arguments.callee(num)
    // إذا كانت الدالة مجهولةً
    countDownFrom(num);
};

// استدعاء الدالة، والتي ستُظهِر الأرقام 5,4,3,2,1,0 على حدة
countDownFrom(5);

</script></body></html>
```

ليس من غير الشائع أن تستدعي الدالة نفسها (أي أنَّ الدالة تعاودية [recursive]).

الفصل الخامس:

الكائن الرئيسي العام

5

1. لمحة نظرية عن مفهوم الكائن الرئيسي

يجب أن تحتوى شيفرة JavaScript نفسها ضمن كائن. فعلى سبيل المثال، عند إنشاء شيفرة JavaScript لبيئة متصفح الويب، فستحتوى JavaScript وتُنفَّذ ضمن كائن window. يُعتَبر الكائن window على أنَّه «كائنٌ رئيسي» (head object)، أو يُشار إليه أحيانًا «بالكائن العام» (the global object). جميع نسخ JavaScript تتطلب استخدام كائن رئيسي وحيد.

يُهيِّئ الكائن الرئيسي من قِبل JavaScript خلف الكواليس لتغليف الشيفرة التي يكتبها المستخدم ولاستضافة الشيفرة التي تأتي مُضمَّنة في JavaScript. تضع JavaScript الشيفرة التي يكتبها المستخدم ضمن الكائن الرئيسي لكي تُنفَّذ. لنتحقق من ذلك فيما يتعلق بمتصفح الويب.

سأُنشِئ في الشيفرة التالية بعض قيم JavaScript وسنتحقق من أنَّ القيم موجودةٌ في الكائن الرئيسي window (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myStringVar = 'myString';
var myFunctionVar = function() {};
myString = 'myString';
myFunction = function() {};

console.log('myStringVar' in window); // الناتج: true
console.log('myFunctionVar' in window); // الناتج: true
console.log('myString' in window); // الناتج: true
console.log('myFunction' in window); // الناتج: true
```

```
</script></body></html>
```

يجب أن تنتبه دومًا أنَّه عندما تكتب شيفرات JavaScript ، فستُكتَب ضمن إطار الكائن الرئيسي. سأفترض في بقية محتوى هذا الفصل أنَّك تعرف أنَّ مصطلح «الكائن الرئيسي» هو مرادف لمصطلح «الكائن العام».

ملاحظة	مجال الكائن الرئيسي هو أكبر مجال (scope) متوافر في بيئة JavaScript.

2. الدوال العامة الموجودة ضمن الكائن الرئيسي

تأتي JavaScript محمّلةً ببعض الدوال المعرَّفة مسبقًا. الدوال الأساسية الآتية تُعتبر دوالًا تابعةً للكائن الرئيسي (مثلًا، يمكننا استخدام الدالة الآتية في متصفح الويب: `window.parseInt('500')`) يمكنك أن تعتبر هذه الدوال أنَّها دوال (تابعة للكائن الرئيسي) جاهزةٌ مباشرةً للاستخدام وتوفرها لغة JavaScript.

- `decodeURI()`
- `decodeURIComponent()`
- `encodeURI()`
- `encodeURIComponent()`
- `eval()`
- `isFinite()`
- `isNaN()`

- `parseFloat()`
- `parseInt()`

3. الكائن الرئيسي والخاصيات والمتغيرات العامة

لا تخلط بين مفهوم الكائن الرئيسي مع مفهوم الخاصيات العامة أو المتغيرات العامة الموجودة في المجال العام (global scope). الكائن الرئيسي هو الكائن الذي يحتوي على جميع الكائنات. أما مصطلح «الخاصيات العامة» أو «المتغيرات العامة» فيُستخدَم للإشارة إلى القيم الموجودة مباشرةً ضمن الكائن الرئيسي والتي لا تكون تابعةً لمجال أحد الكائنات الأخرى. تُعتبَر تلك القيم عامةً لأن جميع الشيفرات بغض النظر عن مكان تنفيذ الشيفرة (من ناحية المجالات تملك وصولًا (عبر scope chain) لتلك الخاصيات والمتغيرات العامة.

سأضع في المثال الآتي الخاصية foo في المجال العام، ثم سأصل إلى تلك الخاصية من مجالٍ آخر (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// إن foo هو متغير عام وهو خاصية للكائن الرئيسي (window)
var foo = 'bar';

// تذكّر أنّ الدوال تُنشِئ مجالًا جديدًا
var myApp = function() {
    var run = function() {
        // الناتج هو bar، وهو قيمة المتغير foo التي عُثِر
        // عليها في الكائن الرئيسي
```

```
        console.log(foo);
    }();
}
myApp();

</script></body></html>
```

أما إذا وضعتُ الخاصية foo خارج المجال العام، فلن تتمكن الدالة console.log من العثور على قيمة foo وستُعيد undefined. وهذا ما هو موضَّح في المثال الآتي:

```
<!DOCTYPE html><html lang="en"><body><script>

// myFunction() إن foo هو متغير موجودٌ في مجال الدالة
var myFunction = function() {var foo = 'bar'};

// تذكّر أنّ الدوال تُنشِئ مجالًا جديدًا
var myApp = function() {
    var run = function() {
        console.log(foo);
    }();
}
myApp();

</script></body></html>
```

وهذا هو السبب وراء إمكانية استدعاء الدوال العامة (مثلًا `window.alert()`) في بيئة المتصفح من أي مجال. ما عليك أن تفهمه هو أنَّ كل شيء في المجال العام سيكون متاحًا لأيّ مجال، وهذا هو سبب تسمية تلك الخاصيات والمتغيرات «بالعامة».

ملاحظة	هنالك فرقٌ ضئيلٌ بين استخدام `var` وعدم استخدام `var` في المجال العام (سينتج عندما «خاصيات عامة» أو «متغيرات عامة»). ألقِ نظرةً على هذا السؤال في StackOverflow للمزيد من التفاصيل.

4. الإشارة إلى الكائن الرئيسي

هنالك عادةً طريقتان للإشارة إلى الكائن الرئيسي. أول طريقة هي الإشارة ببساطة إلى الاسم المعطى للكائن الرئيسي (مثلًا `window` في متصفح الويب). الطريقة الثانية هي استخدام الكلمة المحجوزة `this` في المجال العام. سأستعمل كلا الطريقتين في المثال الآتي (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = 'bar';

windowRef1 = window;
windowRef2 = this;

// window إظهار إشارة (reference) إلى الكائن
console.log(windowRef1, windowRef2);
```

```
// الناتج: 'bar', 'bar'
console.log(windowRef1.foo, windowRef2.foo);

</script></body></html>
```

خزّنا في المثال السابق إشارةً إلى الكائن الرئيسي في متغيرين ثم استخدمناهما للوصول إلى المتغير العام foo.

5. يُستخدَم الكائن الرئيسي ضمنيًا ولا يُشار إليه عادةً بوضوح

لا يُشار عادةً إلى الكائن الرئيسي بوضوح لأنَّه سيشار إليه ضمنيًا. على سبيل المثال، التعبيران البرمجيان window.alert()‎ و alert()‎ متماثلان في بيئة المتصفح؛ حيث تُكمِل JavaScript الفراغات بمفردها. ولأنَّ الكائن window (أي الكائن الرئيسي) هو آخر كائن يتم التحقق من وجود الخاصيات (أو القيم) فيه في «سلسلة المجال» (scope chain)، فسيُشار إلى الكائن الرئيسي ضمنيًا دومًا. سنستعمل في المثال الآتي الدالة alert()‎ الموجودة في المجال العام (مثال حي):

```
<!<!DOCTYPE html><html lang="en"><body><script>

// الكائن window مُشار إليه ضمنيًا هنا (window.foo)
var foo = {
    fooMethod: function() {
        // الكائن window مُشار إليه ضمنيًا هنا (window.alert)
        alert('foo' + 'bar');
        // window مُشار إليه بوضوح هنا وله نفس التأثير
```

```
        window.alert('foo' + 'bar');
    }
}

// الكائن window مُشار إليه ضمنيًا هنا
// window.foo.fooMethod()
foo.fooMethod();

</script></body></html>
```

من المهم أن تفهم أنَّ الكائن الرئيسي مشارٌ إليه ضمنيًا دائمًا، حتى لو لم تُصرِّح بذلك، وهذا لأنَّ الكائن الرئيسي هو آخر كائن في سلسلة المجال (scope chain).

ملاحظة	تحديدك للكائن الرئيسي بوضوح (أي `window.alert()`) بدلًا من `alert()` سيؤدي إلى تقليل الأداء (مدى سرعة تنفيذ الشيفرة) قليلًا فمن الأسرع أن تعتمد على سلسلة المجال (scope chain) بمفردها بدلًا من الإشارة بوضوح إلى الكائن الرئيسي حتى ولو كنتَ تعرف أنَّ ما تريده موجودٌ في المجال العام.

الفصل السادس:

الكلمة المحجوزة this



1. لمحة نظرية عن استخدام this وكيف تُشير إلى الكائنات

عندما تُنشَأ دالةٌ ما، فستُنشَأ كلمةٌ محجوزةٌ (وراء الكواليس) اسمها this، التي تُشير إلى الكائن الذي تعمل عليه الدالة. بعبارةٍ أخرى، المتغير this متاحٌ في مجال الدالة، إلا أنَّه يُشير إلى الكائن الذي تكون تلك الدالة إحدى خاصياته.

لننظر إلى الكائن cody من الفصل الأول مرةً أخرى (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var cody = {
     living : true,
     age : 23,
     gender : 'male',
     getGender : function() {return cody.gender;}
};

console.log(cody.getGender()); // الناتج: 'male'

</script></body></html>
```

لاحظ كيفية وصولنا إلى الخاصية gender داخل الدالة getGender عبر طريقة النقط (مثلًا cody.gender) باستخدام الكائن cody نفسه. يمكن كتابة ما سبق باستخدام this للإشارة إلى الكائن cody، لأنَّ في الواقع this تُشير إلى الكائن cody (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var cody = {
     living : true,
     age : 23,
     gender : 'male',
     getGender : function() {return this.gender;}
};

console.log(cody.getGender()); // الناتج: 'male'

</script></body></html>
```

الكلمة المحجوزة this التي استعملناها في this.gender تُشير ببساطة إلى الكائن cody الذي تعمله عليه الدالة.

ربما يرى البعض أنَّ شرح this مربك ويثير الحيرة، لكن لا تقلق من ذلك. تذكَّر أنَّه عمومًا تُستخدَم الكلمة المحجوزة this داخل الدوال للإشارة إلى الكائن الذي يحتوي على الدالة، بدلًا من الدالة نفسها (الاستثناءات تتضمن استخدام الكلمة المحجوزة new أو call() أو apply()).

ملاحظات	
	- الكلمة المحجوزة this تبدو وتسلك سلوك بقيّة المتغيرات، إلا أنَّك لا تستطيع تعديلها. - على النقيض من arguments والمعاملات التي تُرسَل إلى الدالة، this كلمة محجوزة وليست خاصية.

2. كيف تُحدَّد قيمة this؟

قيمة this المُمرَّرة إلى جميع الدوال تعتمد على السياق الذي تستدعى فيه الدالة في وقت التنفيذ. انتبه جيدًا هنا، لأنَّ هنالك بعض الحالات الشاذة التي عليك تذكُّرها.

أُعطي الكائن myObject في الشيفرة الآتية خاصيةً باسم sayFoo، والتي تُشير إلى الدالة sayFoo. فعندما تُستدعى الدالة sayFoo من المجال العام (global scope)، فستُشير الكلمة المحجوزة this إلى الكائن window، أما عندما تُستدعى كدالة تابعة للكائن myObject، فستشير this إلى myObject.

ولوجود خاصية باسم foo في الكائن myObject، فستُستعمَل قيمة تلك الخاصية (بدلًا من قيمة foo في المجال العام) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = 'foo';

var myObject = {foo: 'I am myObject.foo'};

var sayFoo = function() {
    console.log(this['foo']);
};

// إعطاء الكائن myObject خاصية باسم sayFoo
// وجعلها تشير إلى دالة sayFoo
```

```
myObject.sayFoo = sayFoo;

myObject.sayFoo(); // الناتج: 'I am myObject.foo'

sayFoo(); // الناتج: 'foo'

</script></body></html>
```

من الواضح أنَّ قيمة this تعتمد على سياق استدعاء الدالة. صحيحٌ أنَّ myObject.sayFoo و sayFoo يُشيران إلى نفس الدالة؛ لكن اعتمادًا على مكان (أي سياق) استدعاء الدالة sayFoo()، فستكون قيمة this مختلفة.

هذه نفس الشيفرة السابقة لكن مع التصريح عن استخدام الكائن الرئيسي (أي window) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

window.foo = 'foo';
window.myObject = {foo: 'I am myObject.foo'};

window.sayFoo = function() {
    console.log(this.foo);
};

window.myObject.sayFoo = window.sayFoo;
```

```
window.myObject.sayFoo();
window.sayFoo();

</script></body></html>
```

عليك أن تعي أثناء استخدامك للدوال أو امتلاكك لأكثر من مرجعية إلى دالة معيّنة، أنَّ قيمة this ستتغيّر اعتمادًا على سياق استدعائك للدالة.

> **ملاحظة**
> جميع القيم ما عدا this و arguments تتبع لمكان تعريف الدالة وليس مكان استدعائها (وهذا يُسمى lexical scope).

3. الكلمة المحجوزة this تُشير إلى الكائن الرئيسي في الدوال المتشعبة

ربما تتساءل ما الذي سيحدث للكلمة المحجوزة this عندما تُستعمَل داخل دالة موجودة ضمن دالة أخرى. الأخبار السيئة في ECMA v3 هي أنَّ this تَضلُّ طريقها وتُشير إلى الكائن الرئيسي (الكائن window في المتصفحات) بدلًا من الكائن الذي عُرِّفَت الدالة داخله.

ففي المثال الآتي، ستَضِلُّ الكلمة المحجوزة this الموجودة داخل func2 و func3 طريقها ولن تُشير إلى الكائن myObject وإنما ستُشير إلى الكائن الرئيسي (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>
```

```
var myObject = {
    func1: function() {
        console.log(this); // الناتج: myObject
        var func2 = function() {
            // الناتج هو الكائن window
            // وستشير this إليه من الآن فصاعدًا
            console.log(this);
            var func3 = function() {
                // الناتج هو الكائن window
                // لأنّه هو الكائن الرئيسي
                console.log(this);
            }();
        }();
    }
}

myObject.func1();

</script></body></html>
```

الأخبار الجيدة هي أنَّ هذه المشكلة قد حُلَّت في ECMAScript 5. لكن عليك أن تنتبه لهذه الإشكالية، خصوصًا عندما تبدأ بتمرير الدوال إلى بعضها كقيم.

انظر إلى الشيفرة الآتية وانظر ماذا سيحدث عندما تُمرِّر دالةً مجهولةً إلى الدالة `foo.func1`. فعندما تُستدعى الدالة المجهولة داخل الدالة `foo.func1` (أي دالة داخل دالة)

فستشير this داخل الدالة المجهولة إلى الكائن الرئيسي (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = {
    func1:function(bar) {
        // الناتج هو الكائن window، وليس foo
        bar();

        // ستشير this هنا إلى الكائن foo
        console.log(this);
    }
}

foo.func1(function(){console.log(this)});

</script></body></html>
```

عليك أن تتذكر دومًا أنَّ قيمة this ستُشير دومًا إلى الكائن الرئيسي إن كانت الدالة الموجودة فيها مغلفةً في دالةٍ أخرى أو مستدعاةً في سياق دالةٍ أخرى (وأكرِّر أنَّ الإشكالية السابقة قد حُلَّت في ECMAScript v5).

4. الالتفاف على مشكلة الدوال المتشعبة عبر سلسلة المجال

حسنًا، قيمة this لن تضيع تمامًا، ويمكنك ببساطة توظيف سلسلة المجال (scope chain) لإبقاء إشارة إلى this في الدالة الأب. ستوضِّح الشيفرة الآتية طريقة فعل ذلك عبر استخدام

متغير باسم that (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject = {
    myProperty: 'I can see the light',
    myMethod : function(){
        // (myObject أي) this تخزين إشارة إلى
        // myMethod في مجال الدالة
        var that = this;
        // دالة متشعبة
        var helperFunction = function() {
            // I can see the light الناتج هو
            // that = this لأنّ
            console.log(that.myProperty);
            // الناتج هو كائن window، إن لم نستخدم «that»!
            console.log(this);
        }();
    }
}

myObject.myMethod(); // استدعاء الدالة

</script></body></html>
```

5. التحكم في قيمة this باستخدام call() أو apply()

تُحدَّد قيمة this عادةً من سياق استدعاء الدالة (باستثناء عند استخدام الكلمة المحجوزة new، وسنأتي على ذلك بعد دقائق)، لكن يمكنك التحكم بقيمة this باستخدام apply() أو call() لتحديد ما هو الكائن الذي تُشير الكلمة المحجوزة this إليه عند استدعاء الدالة. استخدام هذه الطرائق يماثل قولنا: «استدعِ الدالة X لكن أخبر الدالة أن تستعمل الكائن Z كقيمة للكلمة المحجوزة this». وبفعلنا لذلك سنتجاوز الطريقة الافتراضية التي تُحدِّد JavaScript فيها قيمة الكلمة المحجوزة this.

سنُنشِئ في المثال الآتي كائنًا ودالةً، ثم سنستدعي الدالة عبر call() لكي نجعل قيمة this داخل الدالة مشيرةً إلى الكائن myObject. ثم ستملأ التعليمات البرمجية داخل الدالة myFunction الكائن myObject بخاصياتٍ معيّنة بدلًا من إسناد تلك الخاصيات إلى الكائن الرئيسي. وبهذا نكون قد عدّلنا الكائن الذي تُشير إليه this (داخل الدالة myFunction) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject = {};

var myFunction = function(param1, param2) {
    // تشير this إلى myObject إذا اُستدعيت الدالة عبر call()
    this.foo = param1;
    this.bar = param2;
    // الناتج: Object {foo = 'foo', bar = 'bar'}
```

```
    console.log(this);
};

// استدعاء الدالة، وضبط قيمة this إلى myObject
myFunction.call(myObject, 'foo', 'bar');

// الناتج: Object {foo = 'foo', bar = 'bar'}
console.log(myObject);

</script></body></html>
```

استعملنا في المثال السابق `call()`، لكن يمكن استخدام `apply()` أيضًا. الفرق بينهما هو في كيفية تمرير المعاملات إلى الدالة. إذ ستُمرَّر المعاملات مفصولةً بفاصلة عند استخدام `call()`؛ أما عند استخدام `apply()`، فستُمرَّر المعاملات داخل مصفوفة. المثال الآتي بنفس فكرة المثال السابق، إلا أنَّه يستعمل `apply()` (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject = {};

var myFunction = function(param1, param2) {
   // تشير this إلى myObject إذا اُستدعيت الدالة عبر apply()
    this.foo = param1;
    this.bar = param2;
    // الناتج: Object {foo = 'foo', bar = 'bar'}
```

```
    console.log(this);
};

// myObject إلى this استدعاء الدالة، وضبط قيمة
myFunction.apply(myObject, ['foo', 'bar']);

// الناتج: Object {foo = 'foo', bar = 'bar'}
console.log(myObject);

</script></body></html>
```

الفكرة التي عليك أن تفهمها هنا هي أنَّك تستطيع أن تتجاوز الطريقة الافتراضية التي تُحدِّد فيها JavaScript قيمة this في مجال الدالة (function scope).

6. استخدام الكلمة المحجوزة this داخل دالة بانية مُعرَّفة من قِبل المستخدم

عندما تُستدعى دالة باستخدام الكلمة المحجوزة new، فقيمة this -داخل الدالة البانية نفسها- تُشير إلى نسخة الكائن التي ستُنشَأ. بعبارةٍ أخرى: في الدالة البانية يمكننا أن نتعامل مع الكائن باستخدام this قبل إنشائه. وفي تلك الحالة ستتغير القيمة الافتراضية للكلمة المحجوزة this بطريقة لا تختلف عن تغيير القيمة باستخدام call() أو apply().

ستُنشِئ في المثال الآتي الدالة البانية Person التي تستخدم this للإشارة إلى كائن يتم إنشاؤه. فعندما تُنشَأ نسخةٌ من الدالة البانية Person، فستشير this.name إلى الكائن الجديد الذي أُنشِئ وتضع فيها خاصيةً اسمها name بقيمةٍ مأخوذةٍ من الوسيط (name) المُمرَّر إلى الدالة

البانية (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var Person = function(name) {
    // ستشير إلى الكائن الجديد الذي سيُنشَأ this
    this.name = name || 'john doe';
}

// إنشاء كائن جديد اعتمادًا على الدالة البانية Person
var cody = new Person('Cody Lindley');

console.log(cody.name); // الناتج: 'Cody Lindley'

</script></body></html>
```

أكرِّر مرةً أخرى أنَّ this تُشير إلى «الكائن الذي سيُنشَأ» عندما تستدعى الدالة البانية باستخدام الكلمة المحجوزة new. أما لو لم نستخدم الكلمة المحجوزة new، فستأخذ this قيمتها من السياق الذي عُرِّفَت فيها الدالة Person، وهو الكائن الرئيسي في هذه الحالة. لننظر إلى مثال (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var Person = function(name) {
    this.name = name || 'john doe';
```

```
}

// new لاحظ أننا لم نستعمل الكلمة المحجوزة
var cody = Person('Cody Lindley');

// هذه القيمة غير معرفة (undefined) وستسبب خطأً
// القيمة الفعلية موجودة في window.name
// console.log(cody.name);

console.log(window.name); // الناتج: 'Cody Lindley'

</script></body></html>
```

7. الكلمة المحجوزة this داخل دالة في الكائن prototype ستُشير إلى الكائن المُنشأ من الدالة البانية

عندما نستعمل this في الدوال المُضافة إلى الخاصية prototype للدالة البانية، فستُشير إلى النسخة من الكائن الذي أُنشِئ من الدالة البانية. فلنقل أنَّ لدينا دالة بانية مخصصة باسم Person()، وتتطلب اسم الشخص الكامل كوسيط. وفي حال أردنا الوصول إلى الاسم الكامل لذاك الشخص، فسنضيف الدالة whatIsMyFullName إلى الخاصية Person.prototype، لذا سترث جميع الكائنات من النوع Person هذه الدالة. وعند استخدامنا للكلمة المحجوزة this في الدالة، فيمكن للدالة أن تصل إلى نسخة الكائن المُنشأة (وبالتالي خاصياتها).

سأشرح عملية إنشاء كائنين من النوع Person (cody و lisa) واللذان سيرثا الدالة

whatIsMyFullName التي تحتوي على الكلمة المحجوزة this للوصول إلى نسخة الكائن (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var Person = function(x){
    if(x){this.fullName = x};
};

Person.prototype.whatIsMyFullName = function(){
    // ستُشير this إلى نسخة الكائن المُنشأة
    // من الدالة البانية Person()
    return this.fullName;
}

var cody = new Person('cody lindley');
var lisa = new Person('lisa lindley');

// استدعاء الدالة الموروثة whatIsMyFullName
// التي تستخدم this للإشارة إلى نسخة الكائن
console.log(cody.whatIsMyFullName(),lisa.whatIsMyFullName());

/*
 ما تزال سلسلة prototype فعّالةً، لذا إن لم يملك أحد الكائنات
 الخاصية fullName، فسيتم البحث عنها في سلسلة prototype.
 سنُضيف أدناه الخاصية fullName إلى الكائن prototype. انظر
```

```
الملاحظة في الأسفل.
*/

Object.prototype.fullName = 'John Doe';
// لم يُمرّر أي وسيط
// لذا لن تُضاف الخاصية fullName إلى أيّة نسخة
var john = new Person();
console.log(john.whatIsMyFullName()); // الناتج: 'John Doe'

</script></body></html>
```

الفكرة التي عليك استيعابها هاهنا هي أنّنا استخدمنا الكلمة المحجوزة this للإشارة إلى نسخ الكائنات عندما وضعناها داخل دالة موجودة في الكائن prototype. وإن لم يملك الكائن خاصيةً ما، فسيتم البحث عنها في سلسلة prototype.

ملاحظة	إذا لم يحتوي الكائن المُشار إليه عبر this خاصيةً مُعيّنة، فستُطبَّق قواعد البحث في سلسلة prototype نفسها. لذا في مثالنا إن لم تكن الخاصية fullName موجودةً في نسخة الكائن فسيتم البحث عنها في Person.prototype.fullName ثم Object.prototype.fullName.

الفصل السابع:

المجالات في JavaScript



1. لمحة نظرية عن المجالات في JavaScript

المجال في JavaScript هو السياق الذي تُنفَّذ فيه الشيفرة، وهنالك ثلاثة أنواع من المجالات: المجال العام (global scope) والمجال المحلي (local scope)، ويُشار إليه أحيانًا بمجال الدالة ([function scope]) ومجال eval.

تكون الشيفرة المُعرَّفة باستخدام var داخلَ دالةٍ موجودةٌ ضمن المجال المحلي، وهي «مرئية» إلى بقية التعبيرات البرمجية في تلك الدالة فقط، بما في ذلك الشيفرات الموجودة ضمن أيّة دوال متشعبة داخلها. أما المتغيرات المُعرَّفة في المجال العام فيمكن الوصول إليها من أي مكان لأنها في أعلى نقطة في سلسلة المجال (scope chain).

تفحّص الشيفرة الآتية بتمعّن وتأكد من أن تفهم أنَّ كل تصريح عن المتغير foo له خصوصيته لأنَّ مجاله يختلف عن البقية (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = 0; // المجال العام
console.log(foo); // 0 الناتج

var myFunction = function() {

    var foo = 1; // المجال المحلي

    console.log(foo); // 1 الناتج
```

```
    var myNestedFunction = function() {
        var foo = 2; // المجال المحلي
        console.log(foo); // الناتج 2
    }();
}();

// eval() مجال الدالة
eval('var foo = 3; console.log(foo);');

</script></body></html>
```

من الضروري أن تفهم أنَّ كل متغير باسم foo يحتوي على قيمة مختلفة لأنَّ كل واحد منهم مُعرَّفٌ في مجالٍ مختلفٍ منفصلٍ

ملاحظات	- يمكن إنشاء عدد لا حصر له من المجالات للدوال ولدالة eval، بينما هنالك مجال عام وحيد مُستخدمٌ من بيئة JavaScript. - المجال العام هو آخر «محطة» في سلسلة المجال. - تُنشِئ الدوال الموجودة ضمن الدوال مجالاتُ تنفيذٍ مُكدَّسة (stacked execution scopes)، ويُشار عادةً إلى المكادس المرتبطة مع بعضها على أنَّها سلسلة المجال (scope chain).

2. لا توجد مجالات كتلية في JavaScript

لا توجد مجالاتٌ كتليةٌ لأنَّ التعابير المنطقية (مثل if) وحلقات التكرار (مثل for) لا تُنشِئ

مجالًا كتليًا (block scope) خاصًا بها، لذا يمكن أن تُعاد الكتابة فوق المتغيرات. أمعن النظر في الشيفرة الآتية وتأكّد أنّك تفهم لماذا سيُعاد تعريف قيمة المتغير foo أثناء تنفيذ الشيفرة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = 1; // foo = 1

if (true) {
    foo = 2; // foo = 2
    for(var i = 3; i <= 5; i++) {
        foo = i; // foo = 3,4,5
        console.log(foo); // الناتج: 3,4,5
    }
}

</script></body></html>
```

إذًا تتغير قيمة foo أثناء تنفيذ الشيفرة لأنّ JavaScript لا تملك مجالًا كتليًا، وإنما هنالك مجالٌ عامٌ ومجالٌ محلي (تابعٌ للدالة)، ومجالٌ تابعٌ لدالة eval.

3. استخدام var داخل الدوال للتصريح عن المتغيرات ولتفادي التصادم بين المجالات

ستُعرِّف JavaScript أيّة متغيرات لا يسبقها var (حتى تلك الموجودة في دالة أو سلسلة من

الدوال) على أنها في المجال العام بدلًا من المجال المحلي. ألقِ نظرة إلى الشيفرة الآتية ولاحظ أنَّه دون استخدام var للتصريح عن المتغير bar فإن القيمة ستُعرَّف في المجال العام وليس المجال المحلي (مكان تعريفها الافتراضي) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = function() {
    var boo = function() {
        // var لم نستعمل
        // window.bar في المجال العام في سيوضع bar لذا
        bar = 2;
    }();
}();

// الناتج 2، لأن المتغير bar موجودٌ في المجال العام
console.log(bar);

// … وذلك على النقيض من

var foo = function() {
    var boo = function() {
        var doo = 2;
    }();
}();
```

```
// الناتج undefined، لأنّ doo في مجال الدالة boo، وسيحدث خطأ
// console.log(doo);

</script></body></html>
```

الفكرة هنا هي أنَّه عليك استخدام var دائمًا عند تعريف المتغيرات داخل الدوال، وهذا سيجنِّبُك التعامل مع مشاكل مُحيّرة ومُربِكة تتبع للمجالات. هنالك استثناءٌ لهذه القاعدة هو عندما تريد أن تُنشِئ أو تُعدِّل الخاصيات في المجال العام من داخل دالة عمدًا.

4. سلسلة المجال

هنالك سلسلةٌ من عمليات البحث التي تحدث عندما تبحث JavaScript عن قيمةٍ مرتبطةٍ بمتغير. هذه السلسلة مبنية على هيكلية المجال (hierarchy of scope). ففي الشيفرة الآتية، سأعرض قيمة sayHiText من داخل مجال الدالة func2 (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var sayHiText = 'howdy';

var func1 = function() {
    var func2 = function() {
        // نحن في مجال func2
        // لكن سيتم العثور على sayHiText في المجال العام
        console.log(sayHiText);
    }();
```

```
}();

</script></body></html>
```

كيف يمكن العثور على قيمة `sayHiText` إن لم تكن موجودةً في مجال الدالة `func2`؟ ستبحث JavaScript أولًا في الدالة `func2` عن المتغير المسمى `sayHiText`، ولن تعثر عليه هنالك، ومن ثم ستبحث في الدالة «الأب» (parent function) المسماة `func1`، ولن تعثر على المتغير `sayHiText` في مجال الدالة `func1` أيضًا، لذا ستكمل JavaScript بحثها لتصل إلى المجال العام حيث ستعثر على `sayHiText`، وعندها ستحصل على قيمة المتغير `sayHiText`؛ وإن لم يكن المتغير `sayHiText` معرفًا في المجال العام فسيتم إعادة القيمة `undefined` من قِبل JavaScript.

من المهم جدًا أن تستوعب هذا المفهوم؛ لننظر إلى شيفرةٍ أخرى التي سنعرض فيها ثلاث قيم من ثلاثة مجالات مختلفة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var x = 10;
var foo = function() {
    var y = 20;
    var bar = function() {
        var z = 30;
        // المتغير z محلي، والمتغيران y و x سيُعثَر
        // عليهما في سلسلة المجال
```

```
        console.log(z + y + x);
    }();
};

foo(); // الناتج: 60

</script></body></html>
```

قيمة المتغير z هي قيمةٌ محلية في الدالة bar وهي في نفس مجال استدعاء الدالة console.log، أما القيمة y فهي في مجال الدالة foo، التي هي الدالة «الأب» للدالة ()bar، وقيمة x موجودة في المجال العام. وجميع تلك المتغيرات متاحةٌ للوصول من داخل الدالة bar عبر سلسلة المجال. من الضروري أن تفهم أنَّ المتغيرات المذكورة في الدالة bar سيتم البحث عنها عبر سلسلة المجال للحصول على قيمها.

ملاحظة	إن فكرتَ في الأمر مليًا، فستجد أنَّ سلسلة المجال (scope chain) لا تختلف كثيرًا عن سلسلة prototype (prototype chain). وكلاهما عبارةٌ عن طريقةٍ للبحث عن قيمةٍ في أماكن مُحدَّدة تنظيميًا وهيكليًا.

5. ستُعيد سلسلة المجال أول قيمة يُعثَر عليها

هنالك متغير في الشيفرة الآتية اسمه x موجودٌ في نفس المجال الذي ستُنفَّذ فيه الدالة console.log، وستُستعمَل القيمة «المحلية» للمتغير x، والبعض يعتبر أنَّ القيمة المحلي ستخفي أو تغطي أو تضع قناعًا على المتغيرات التي لها نفس الاسم (أي x) في سلسلة المجال

(مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var x = false;
var foo = function() {
    var x = false;
    bar = function() {
        var x = true;
        /*
 المتغير المحلي x هو أول متغير سيُعثَر عليه في السلسلة، لذا
سيُغطي على غيره من المتغيرات التي تحمل نفس الاسم لكن في مجالٍ
        آخر
        */
        console.log(x);
    }();
}

foo(); // الناتج: true

</script></body></html>
```

تذكر أنَّ البحث عن قيمةٍ للمتغير في سلسلة المجال سيتوقف عند أقرب نقطة يُعثَر فيها على قيمة للمتغير، حتى لو وجِدَ متغيرٌ بنفس الاسم في محطاتٍ أبعد في السلسلة.

6. سيُحدَّد المجال أثناء تعريف الدالة وليس عند استدعائها

لمّا كانت هنالك مجالاتٌ محليّة تُحدَّد عبر الدوال، ويمكن في الوقت نفسه تمرير الدوال إلى بعضها كغيرها من القيم في JavaScript، فربما يظن بعضنا أنَّ فك تشفير سلسلة المجال في هذه الحالة هو أمرٌ معقدٌ جدًا، لكنه سهلٌ جدًا في الواقع. تُحدَّد سلسلة المجال بناءً على مكان الدالة أثناء تعريفها، وليس عند استدعائها. ويسمى هذا أيضًا «lexical scoping». فكِّر مليًا بهذا الأمر، لأنَّ أغلبية الأشخاص يتعثرون به كثيرًا في شيفرات JavaScript.

ستُنشَأ سلسلة المجال قبل استدعاء الدالة؛ وبسبب ذلك فيمكننا إنشاء «تعابير مغلقة» (closures). فعلى سبيل المثال، يمكننا إنشاء دالة تُعيد دالة مُتشعبة فيها إلى المجال العام، وبالتالي ستتمكن الدالة المتشعبة من الوصول -عبر سلسلة المجال- إلى مجال الدالة الأب.

سنُعرِّف في المثال الآتي الدالة `parentFunction` التي تُعيد دالةً مجهولةً، ومن ثم سنستدعي الدالة المُعادة في المجال العام. ولأنَّ الدالة المجهولة مُعرَّفة وموجودة داخل `parentFunction`، فما زالت تملك وصولًا إلى مجال `parentFunction` عند استدعائها، وهذا ما نسميه «بالتعابير البرمجية المغلقة» (closures) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var parentFunction = function() {
    var foo = 'foo';
    // إعادة دالة مجهولة
    return function() {
        console.log(foo); // الناتج: 'foo'
```

```
    }
}

// المتغير nastedFunction يُشير إلى الدالة
// المتشعبة المُعادة من parentFunction
var nestedFunction = parentFunction();

// الناتج foo، لأن الدالة المُعادة
// ستتمكن من الوصول إلى foo عبر سلسلة المجال
nestedFunction();

</script></body></html>
```

عليك أن تفهم هنا أنَّ سلسلة المجال تُحدَّد أثناء التعريف (أي كما كُتِبَت الشيفرة تمامًا). وتمرير الدوال إلى بعضها داخل الشيفرة الخاصة بك لن يُغيّر شيئًا في سلسلة المجال.

7. التعابير المغلقة سببها هو سلسلة المجال

ضع بذهنك كل ما تعلمتَه عن سلسلة المجال وكيفية البحث فيها في هذا الفصل، ولن يتعسّر عليك فهم التعابير المغلقة (closures). سأُنشِئ في المثال الآتي دالةً باسم `countUpFromZero`، وهذه الدالةُ تُعيد دالةً موجودةً ضمنها، فعندما تُستدعى الدالة المتشعبة فستملك وصولًا إلى مجال الدالة الأب بسبب سلسلة المجال (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>
```

```
var countUpFromZero = function() {
    var count = 0;
    // countUpFromZero تُعيد الدالة المتشعبة عند استدعاء
    return function() {
        // المتغير count مُعرّف في الدالة الأب
        return ++count;
    };
}(); // ستُستدعى مباشرةً، وستُعاد الدالة المتشعبة

console.log(countUpFromZero()); // الناتج: 1
console.log(countUpFromZero()); // الناتج: 2
console.log(countUpFromZero()); // الناتج: 3

</script></body></html>
```

في كل مرة تُستدعى فيها الدالة countUpFromZero، فستملك الدالة المجهولة الموجودة في (والمُعادة من) الدالة countUpFromZero وصولًا إلى مجال الدالة الأب. وهذه التقنية التي تعتمد على طريقة عمل سلسلة المجال هي مثالٌ عن تعبيرٍ برمجيٍ مغلق (closure).

ملاحظة	إن أحسستَ أنني أبسّط التعابير المغلقة أكثر من اللازم، فأنت مصيبٌ في ذلك. لكنني فعلتُ هذا عن قصد لأنني أعتقد أنَّ المهم هو ترسيخ الفهم الصحيح للدوال والمجالات، وليس الدخول في تعقيدات غير ضرورية تأتي في سياق التنفيذ. إن احتجتَ إلى شرحٍ تفصيليٍ عن التعابير المغلقة، فأنصحك بالنظر إلى مقالة «JavaScript Closures».

الفصل الثامن:

خاصية prototype التابعة للدوال



1. لمحة نظرية عن سلسلة prototype

الخاصية `prototype` هي كائنٌ مُنشأٌ من JavaScript لكل نسخةٍ من الكائن `Function()`. وإذا ابغتينا الدقة، فإنها تربط بين نسخ الكائنات المُنشَأة باستخدام الكلمة المحجوزة `new` إلى الدالة البانية التي أنشأتها. والسبب وراء فعل ذلك هو السماح للكائنات بمشاركة -أو وراثة- الدوال والخاصيات الشائعة المشتركة بينهم. وأهم ما في الأمر أنَّ المشاركة تحدث أثناء البحث عن قيمةِ خاصيةٍ ما. تذكَّر من الفصل الأول أنَّه في كل مرة تبحث أو تحاول الوصول فيها إلى خاصية في كائن، فسيتم البحث عنها في خاصيات الكائن فإن لم يُعثَر عليها فسيتكمل البحث في سلسلة prototype.

ملاحظة	يُنشَأ الكائن prototype (prototype object) لكل دالة، بغض النظر عمّا إذا كنتَ تريد استخدام الدالة كدالة بانية أم لا.

سأُنشِئ في المثال الآتي مصفوفةً من الدالة البانية `Array()`، ثم سأستدعي الدالة `join()` (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = new Array('foo', 'bar');

console.log(myArray.join()); // الناتج: 'foo,bar'

</script></body></html>
```

الدالة `join()` غير مُعرَّفة كخاصية في نسخة الكائن `myArray`، لكننا تمكننا من الوصول إليها بطريقةٍ ما كما لو كانت مُعرَّفةً وتابعةً لذاك الكائن. من المؤكد أنَّ هذه الدالة مُعرفَّة بمكانٍ ما، لكن أين؟ حسنًا، هي مُعرَّفة كخاصية تابعة لخاصية `prototype` للدالة البانية `Array()`. ولأن لغة JavaScript لم تجد الدالة `join()` ضمن الكائن `myArray`، فستبحث JavaScript عبر سلسلة prototype عن دالةٍ باسم `join()`.

حسنًا، ما السبب وراء فعلنا لذلك بهذه الطريقة؟ السبب الحقيقي هو الكفاءة وقابلية إعادة الاستخدام. لماذا يجب على كل كائن يُمثِّل مصفوفة تم إنشاؤه من الدالة البانية `Array()` أن يُعرِّف دالة `join()` خاصة به في حين أنَّ الدالة `join()` ستعمل بشكل متماثل في جميع تلك الكائنات؟ من المنطقي أن تتشارك جميع المصفوفات بنفس دالة `join()` دون الحاجة إلى إنشاء نسخة من الدالة لكل كائن يُمثِّل مصفوفةً.

تمكننا من تحقيق الكفاءة التي تحدثنا عنها بفضل خاصية `prototype` وربط prototype (أي prototype linkage) والبحث في سلسلة prototype. وسنشرح في هذا الفصل خصوصيات الوارثة من الكائن prototype التي تُسبِّب عادةً ارتباكًا لِمنَ لا يألفها. لكن من الأفضل لك أن تبدأ بتذكر آلية عمل سلسلة prototype. ارجع إلى الفصل الأول إذا احتجتَ إلى تذكرة عن كيفية تحديد قيم الخاصيات.

2. لماذا علينا أن نهتم بخاصية prototype؟

علينا أن نهتم بخاصية `prototype` لأربعة أسباب.

أ. السبب الأول

أول سبب هو أنَّ خاصية `prototype` مُستعملة من قِبل الدوال البانية الموجودة في أساس اللغة (مثلًا `Object()` و `Array()` و `Function()` ...إلخ.) للسماح للكائنات المُنشأة من تلك الدوال البانية أن ترث خاصياتٍ ودوالًا وهي الآلية التي تستخدمهاJavaScript نفسها للسماح للكائنات بأن ترث الخاصيات والدوال من خاصية `prototype` للدالة البانية. إذا أردتَ أن تفهم JavaScript فهمًا تامًا، فعليك أن تفهم كيف تستعمل JavaScript الكائن `prototype`.

ب. السبب الثاني

عند إنشائك للدوال البانية المُعرَّفة من قِبلك، فيمكنك أن تستعمل أساليب الوراثة التي تستعملها الكائنات الأساسية في لغة JavaScript. لكن عليك أولًا أن تفهم كيفية الاستفادة من الخاصية `prototype`.

ت. السبب الثالث

ربما لا تحب الوراثة من الكائن `prototype` أو تُفضِّل نمطًا آخر من وراثة الكائنات، لكن واقعيًا قد تضطر في أحد الأيام أن تُعدِّل أو تدير شيفرةً كتبها مبرمجٌ آخر الذي يستخدم الوراثة عبر الكائن `prototype` كثيرًا؛ وعندما يحدث ذلك فعليك أن تكون محيطًا بآلية عمل الوراثة عبر الكائن `prototype`، بالإضافة إلى كيفية استعمال المطورين لهذا الكائن عندما يُنشؤون دوالًا بانيةً خاصةً بهم.

ث. السبب الرابع

عند استخدام الوراثة عبر الكائن `prototype`، فستتمكن من إنشاء كائنات التي تتشارك

بنفس الدوال. وكما ذكرتُ سابقًا، لا تحتاج جميع المصفوفات -التي هي نسخٌ منشأةٌ من الدالة البانية Array()- أن تُعرِّف دالة join() خاصة بها، فجميع النسخ تستطيع استخدام دالة join() نفسها لأنَّ الدالة موجودة في سلسلة prototype.

3. الخاصية prototype موجودة في جميع الدوال

منشأ جميع الدوال هو الدالة البانية Function()، وحتى لو لم تستدعي الدالة البانية Function() مباشرةً (مثلًا var add = new Function('x', 'y', 'return x + y');) واستخدمتَ الشكل المختصر (مثلًا var add = function(x, y) {return x + y};).

عند إنشاء دالة فستعطى دائمًا خاصية prototype والتي هي كائنٌ فارغ. سنُعرِّف في المثال الآتي دالةً باسم myFunction، ثم سنحاول الوصول إلى الخاصية prototype، والتي هي كائنٌ فارغٌ (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myFunction = function() {};
console.log(myFunction.prototype); // الناتج: object{}
console.log(typeof myFunction.prototype); // الناتج: 'object'

</script></body></html>
```

تأكّد أنّك تفهم تمامًا أنّ خاصية prototype تأتي من الدالة البانية Function(). لن

تُستخدَم الخاصية prototype في دوالنا إلا إذا استخدمناها كدوال بانية، لكن هذا لا يغيّر حقيقة إعطاء الدالة البانية `Function()` كل كائن مُنشَأ منها الخاصيةَ prototype.

4. الخاصية prototype الافتراضية هي كائن Object()

أعلمُ أنَّ حديثنا عن خاصية prototype أصبح ثقيلًا ومعقدًا. لكن حقيقةً، خاصية prototype هي كائنٌ فارغٌ اسمه «prototype» تُنشِئه JavaScript وراء الكواليس وتوفِّره عند استدعاء الدالة البانية `Function()`. إذا أردتَ أن تُجري العملية يدويًا، فيمكنك تنفيذ شيءٍ شبيهٍ بما يلي (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myFunction = function() {};
// إضافة الخاصية prototype وضبط قيمتها إلى كائنٍ فارغ
myFunction.prototype = {};
// الناتج هو كائنٌ فارغ
console.log(myFunction.prototype);

</script></body></html>
```

في الواقع، الشيفرة السابقة سليمة تمامًا، وهي تفعل أمرًا شبيهًا بما فعلته JavaScript.

ملاحظة	يمكن أن تُضبَط قيمة الخاصية prototype إلى أيّة قيمة معقدة (أي كائن متوافرة في لغة JavaScript. ستتجاهل JavaScript أيّة خاصيات prototype مضبوطةٌ قيمتها إلى قيمةٍ أوليّة.

5. النسخ المُنشَأة من الدالة البانية مربوطةٌ بخاصية prototype التابعة للدالة البانية

صحيحٌ أنَّ الخاصية prototype هي مجرد كائن، لكن لها خصوصيتها بأنَّ سلسلة prototype تربط كل نسخة كائن إلى الخاصية prototype التابعة للدالة البانية. وهذا يعني أنَّه في أي وقت يُنشَأ فيه كائنٌ من الدالة البانية باستخدام الكلمة المحجوزة new (أو عندما يُنشَأ كائن لاحتواء قيمة أولية)، فسيُضاف رابطٌ خفيٌ بين نسخة الكائن المُنشأ وخاصية prototype الخاصة بالدالة البانية التي اُستخدِمَت لإنشائها؛ يُعرَف هذا الرابط داخل نسخة الكائن على أنَّه `__proto__`. تَربطُ JavaScript الكائنات بخاصية prototype في الخلفية أثناء استدعاء الدالة البانية، وفي الواقع هذا الرابط هو الذي يسمح بأن تكون سلسلة prototype «سلسلةً»!

سنضيف في المثال الآتي خاصيةً إلى الخاصية prototype التابعة للدالة البانية `Array()`، والتي سنتمكن من الوصول إليها من كائنٍ من النوع `Array()` عبر خاصية `__proto__` الموجودة في نسخة الكائن (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// هذه الشيفرة ستعمل فقط في المتصفحات
// التي تدعم الوصول إلى __proto__

Array.prototype.foo = 'foo';
var myArray = new Array();

// الناتج foo، لأن myArray.__proto__ = Array.prototype
```

```
console.log(myArray.__proto__.foo);

</script></body></html>
```

ولمّا كانت إمكانية الوصول إلى الخاصية __proto__ ليست جزءًا من معيار ECMA قبل الإصدار السادس (لكن أضيفَت إلى ECMAScript الإصدار السادس) ، فهنالك طريقةٌ أكثر شمولًا للحصول على رابط إلى كائن prototype الخاص به، وذلك عبر استخدام الخاصية constructor. وهذا ما هو موضَّح في المثال الآتي (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// foo جميع نسخ كائنات Array() سترث الخاصية
Array.prototype.foo = 'foo';
var myArray = new Array();

// تَتَبُّع الخاصية foo بالطريقة العامة عبر استخدام
// *.constructor.prototype

// الناتج: foo
console.log(myArray.constructor.prototype.foo);

// أو يمكننا -بكل تأكيد- الاستفادة من سلسلة prototype
console.log(myArray.foo) // الناتج: foo
// سنستخدم سلسلة prototype للعثور على الخاصية في
// Array.prototype.foo
```

```
</script></body></html>
```

الخاصية foo في الشيفرة السابقة موجودة ضمن كائن prototype. عليك أن تُدرِك أنَّ الوصول إلى الخاصية foo أصبح ممكنًا لوجود رابط بين نسخة الكائن من النوع Array() وبين كائن prototype التابع للدالة البانية Array() (أي Array.prototype). باختصار: myArray.__proto__ (أو myArray.constructor.prototype) تُشير إلى Array.prototype.

6. آخر محطة في سلسلة prototype هي Object.prototype

لمّا كانت الخاصية prototype عبارةً عن كائنٍ، فإن آخر محطة في سلسلة prototype هي Object.prototype. ستُنشِئ في الشيفرة الآتية الكائن myArray، والذي هو مصفوفةٌ فارغةٌ؛ ثم سنحاول الوصول إلى خاصيةٍ للكائن myArray لكنها غير معرفة بعد، مما يؤدي إلى البحث عنها في سلسلة prototype. سيبدأ البحث عن الخاصية foo في الكائن myArray، ولعدم وجودها فيه فسيستمر البحث عن الخاصية في Array.prototype، ولعدم وجودها فيه أيضًا، فسيتم البحث في آخر مكان ألا وهو Object.prototype. ولعدم تعريف الخاصية foo في الكائنات الثلاثة السابقة، فستُعاد القيمة undefined (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = [];
console.log(myArray.foo) // الناتج: undefined
```

```
/*
لم يُعثَر على قيمة للخاصية foo في myArray.foo أو
 Array.prototype.foo أو Object.prototype.foo، لذا فإنّ قيمتها
هي undefined.
*/

</script></body></html>
```

لاحظ أنَّ البحث في السلسلة توقّف عند `Object.prototype`.

> **تحذير**
> انتبه! أي شيء يُضاف إلى الكائن `Object.prototype` سيظهر في حلقة `for in`.

7. سلسلة prototype ستُعيد أول خاصية يُعثَر عليها في السلسلة

كما في سلسلة المجال، فإن سلسلة prototype ستستخدم أول قيمة تجدها في سلسلة البحث.

لو عدَّلنا شيفرة المثال السابق، وأضفنا نفس الخاصية إلى الكائنين `Object.prototype` و `Array.prototype`، فعندما نحاول الوصول إلى قيمة الخاصية `foo` في كائنٍ من نوع `Array()` فستُعاد القيمة الموجودة في كائن `Array.prototype` (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>
```

```
Object.prototype.foo = 'object-foo';
Array.prototype.foo = 'array-foo';
var myArray = [];

// الناتج array-foo الموجود في Array.prototype.foo
console.log(myArray.foo);

myArray.foo = 'bar';

// الناتج bar الموجود في myArray.foo
console.log(myArray.foo);

</script></body></html>
```

في الشيفرة السابقة، ستُغطي قيمة foo الموجودة في `Array.prototype.foo` على قيمة foo الموجودة في `Object.prototype.foo`. تذكَّر أنَّ عملية البحث في السلسلة ستنتهي عندما يُعثَر على قيمةٍ ما في السلسلة، حتى ولو كانت هنالك خاصيةٌ لها نفس الاسم في «محطةٍ» أبعد في السلسلة.

8. تبديل خاصية prototype ضمن كائنٍ جديدٍ سيؤدي إلى حذف خاصية constructor الافتراضية

من الممكن استبدال القيمة الافتراضية للخاصية `prototype` ووضع قيمة جديدة مكانها؛ لكنَّ فعل ذلك سيؤدي إلى حذف الخاصية `constructor` الموجودة في كائن `prototype`

الأصلي - إلا إذا حدَّدتها بعد ذلك يدويًا.

أنشأنا في الشيفرة الآتية الدالة البانية Foo، ثم وضعنا كائنًا فارغًا قيمةً للخاصية prototype، ثم تحققنا من حدوث خلل في خاصية constructor (أصبحت تُشير إلى Object) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var Foo = function Foo(){};

// وضع كائن فارغ في خاصية prototype
Foo.prototype = {};

var FooInstance = new Foo();

// الناتج false، لأننا «كسرنا» المرجعية إليها
console.log(FooInstance.constructor === Foo);
// الناتج Object()، وليس Foo()
console.log(FooInstance.constructor);

// قارن ما سبق بالشيفرة الآتية التي لم نخرِّب فيها
// قيمة الخاصية prototype

var Bar = function Bar(){};

var BarInstance = new Bar();
```

```
console.log(BarInstance.constructor === Bar); // الناتج: true
console.log(BarInstance.constructor); // الناتج: Bar()

</script></body></html>
```

إذا كنتَ تنوي تبديل قيمة خاصية prototype الافتراضية (وهذا شائعٌ في بعض أنماط التصميم في البرمجة كائنية التوجه [OOP] في JavaScript) المضبوطة من لغة JavaScript، فيجب عليك أن تُعيد ربط خاصية constructor التي تُشير إلى الدالة البانية. ستُغيّر في الشيفرة السابقة لكي تُشير الخاصية constructor مرةً أخرى إلى الدالة البانية الصحيحة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var Foo = function Foo(){};

Foo.prototype = {constructor:Foo};

var FooInstance = new Foo();

console.log(FooInstance.constructor === Foo); // الناتج: true
console.log(FooInstance.constructor); // الناتج: Foo()

</script></body></html>
```

9. الكائنات التي ترث خاصيات من prototype ستحصل دومًا على أحدث القيم

الخاصية prototype ديناميكية في أنَّ جميع نسخ الكائنات ستحصل على أحدث قيمة، بغض النظر عن مكان تعريفها أو تغيرها أو إسناد قيمةٍ إليها. سنُنشِئ في الشيفرة الآتية الدالة البانية Foo، ثم سنضيف الخاصية x إلى كائن prototype ثم سنُنشِئ نسخةً من Foo() باسم FooInstance ثم سنعرض قيمة x ثم سنُحدِّث قيمة x الموجودة في الكائن prototype ثم سنعرضها مرةً أخرى، وسنجد أنَّ الكائن يستطيع الوصول إلى أحدث قيمة موجودة في كائن prototype (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var Foo = function Foo(){};

Foo.prototype.x = 1;

var FooInstance = new Foo();

// الناتج 1
console.log(FooInstance.x);

Foo.prototype.x = 2;

// الناتج 2، لاحظ كيف تحدث القيمة تلقائيًا
```

```
console.log(FooInstance.x);

</script></body></html>
```

أظن أنَّ هذا السلوك لن يُفاجئك لأنك تعلم كيف تعمل سلسلة البحث عن المتغيرات. وسيعمل ما سبق عملًا صحيحًا بغض النظر عمّا إذا كنتَ تستعمل الكائن prototype الأصلي أم أنَّك استبدلته واستخدمتَ كائنًا خاصًا بك عوضًا عنه. سأبدِّل قيمة الكائن prototype الافتراضية في المثال الآتي لشرح هذه الفكرة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var Foo = function Foo(){};

// ستظهر نفس النتائج كما في المثال السابق
Foo.prototype = {x:1};

var FooInstance = new Foo();

// الناتج 1
console.log(FooInstance.x);

Foo.prototype.x = 2;

// الناتج 2، لاحظ كيف تحدثت القيمة تلقائيًا
console.log(FooInstance.x);
```

```
</script></body></html>
```

10. تغيير قيمة prototype إلى كائنٍ جديد لن يؤدي إلى تحديث النسخ المُنشَأة سابقًا

ربما تظن أنَّك تستطيع استبدال خاصية prototype كليًّا في أيّ وقت وستُحدَّث جميع نسخ الكائنات، لكن هذا ليس صحيحًا. فعندما تُنشِئ نسخةً من كائنٍ، فسترتبط بقيمة prototype الموجودة أثناء تهيئة الكائن. وضع كائن جديد بدلًا من قيمة الخاصية prototype الأصلية لن يُحدِّث الارتباط بين النسخ المُنشَأة مسبقًا من الكائن وبين قيمة خاصية prototype الجديدة. لكن تذكَّر -كما ذكرتُ سابقًا- أنَّك تستطيع تحديث أو إضافة الخاصيات إلى الكائن prototype الأصلي وستبقى تلك القيم «متصلةً» ومتاحةً إلى النسخ المُنشأة مسبقًا من الكائن (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var Foo = function Foo(){};

Foo.prototype.x = 1;

var FooInstance = new Foo();

// الناتج 1، كما قد تتوقع
console.log(FooInstance.x);
```

```
// prototype لنحاول الآن استبدال قيمة الكائن
// ووضع كائن Object() جديد بدلًا منه
Foo.prototype = {x:2};

// الناتج 1، ماذا؟! ألا يجب أن تُعرَض القيمة 2؟
// ألم نُحدِّث الكائن prototype منذ قليل؟
console.log(FooInstance.x);
/*
سيشير FooInstance إلى كائن prototype الذي كان موجودًا أثناء تهيئته.
*/

// إنشاء كائن جديد من الدالة البانية Foo()
var NewFooInstance = new Foo();

// يجب أن ترتبط النسخة الجديدة إلى قيمة
// الكائن prototype الجديدة (أي {x:2};)
console.log(NewFooInstance.x); // الناتج: 2

</script></body></html>
```

الفكرة هنا أنَّ عليك عدم استبدال الكائن prototype ووضع كائن جديد مكانه بعد أن تبدأ بإنشاء الكائنات. وفعلك لذلك سيؤدي إلى وجود نسخٍ تُربَط بكائناتِ prototype مختلفةٍ.

11. يمكن للدوال البانية المُعرَّفة من المستخدم استخدام الوراثة من الكائن prototype كما في الدوال البانية الأساسية

أأمل أن تكون في هذه المرحلة من هذا الفصل قد فهمتَ كيف تستفيد JavaScript نفسها من الخاصية `prototype` للوراثة (مثلًا `Array.prototype`). يمكن استخدام نفس الطريقة لإنشاء دوال بانية غير مُضمّنة في أساس اللغة (أي مُعرَّفة من المستخدم). سنحاكي في المثال الآتي النمط الذي تستخدمه JavaScript للوراثة في كائن `Person` الذي نألَفُهُ (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var Person = function() {};

// countLimbs و arms و legs جميع نسخ Person سترث الخاصيات
Person.prototype.legs = 2;
Person.prototype.arms = 2;
Person.prototype.countLimbs = function() {return this.legs +
this.arms;};

var chuck = new Person();

console.log(chuck.countLimbs()); // الناتج: 4

</script></body></html>
```

أنشأنا في بداية المثال السابق الدالة البانية `Person()`، ثم أضفنا بعض الخاصيات إلى

الخاصية `prototype` التابعة للدالة البانية `Person()`، والتي سترثها جميع الكائنات. لذا ستتمكن بكل بساطة من استخدام سلسلة prototype بنفس الطريقة التي تستعملها JavaScript فيها للوراثة في الكائنات الأساسية (المُضمَّنة في اللغة).

أحد الأمثلة العملية عن كيفية استفادتك من هذا هو إنشاء دالة بانية التي ترث فيها الكائنات المُنشأة منها الخاصيتين `legs` و `arms` إن لم تُمرَّرا كمعاملَين. ففي المثال الآتي، إن أُرسِلَت المعاملات إلى الدالة البانية `Person()` فستُستخدَم كخاصيات تابعة للكائن، أما إذا لم يُرسَل أحدهما أو كلاهما، فهنالك بديلٌ عنهما. الخاصيات التابعة للكائن (إن مُرِّرَت عبر الدالة البانية) ستغطي على الخاصيات الموروثة، وبهذا ستحصل على أفضل ما في الطريقتين (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var Person = function(legs, arms) {
    // التغطية على القيم الموجودة في prototype
    if (legs !== undefined) {this.legs = legs;}
    if (arms !== undefined) {this.arms = arms;}
};

Person.prototype.legs = 2;
Person.prototype.arms = 2;
Person.prototype.countLimbs = function() {return this.legs +
this.arms;};

var chuck = new Person(0, 0);
```

```
console.log(chuck.countLimbs()); // الناتج: 0

</script></body></html>
```

12. إنشاء سلاسل وراثة

الغرض من الوراثة من الكائن prototype هو السماح بوجود سلاسل وراثة (inheritance chains) التي تُحاكي أنماط الوراثة الموجودة في لغات البرمجة التي تُطبِّق مفاهيم البرمجة كائنية التوجه (object oriented programming) التقليدية. فلكي يرث أحد الكائنات من كائنٍ آخر في JavaScript فكل ما عليك فعله هو إنشاء نسخة من الكائن الذي تريد الوراثة منه وإسناده كقيمةٍ لخاصية prototype للدالة التي تُنشِئ الكائنات التي تريدها أن ترث تلك الخاصيات.

في الشيفرة الآتية، سترث كائنات Chef (أي النسخة cody) الخاصيات من Person() وهذا يعني أنَّه لم يُعثَر على خاصيةٍ ما في الكائن Chef فسيُبحَث عنها في كائن prototype في الدالة التي تُنشِئ كائنات Person(). ولكي تربط عملية الوراثة فعليك أن تُهيِّئ نسخةً من Person() وتُسندها كقيمة إلى خاصية Chef.prototype (أي: Chef.prototype = new Person();) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var Person = function(){this.bar = 'bar'};
```

```
Person.prototype.foo = 'foo';

var Chef = function(){this.goo = 'goo'};
Chef.prototype = new Person();

var cody = new Chef();

console.log(cody.foo); // الناتج: 'foo'
console.log(cody.goo); // الناتج: 'goo'
console.log(cody.bar); // الناتج: 'bar'

</script></body></html>
```

كل ما فعلناه في المثال السابق هو استخدام النظام الذي تستعمله الكائنات المُضمنة في اللغة. اعتبر أنَّ `Person()` لا تختلف عن القيمة `Object()` الافتراضية لخاصية `prototype`. بعبارةٍ أخرى، هذا هو ما يحدث تحديدًا عندما تبحث الخاصية `prototype` -التي تحتوي قيمتها الافتراضية الفارغة القيمةَ `Object()`- في خاصية `prototype` للدالة البانية التي أنشأتها (أي `Object.prototype`) عن الخاصيات التي يجب أن ترثها.

الفصل التاسع:

المصفوفات والكائن ()Array

9

1. لمحة نظرية عن استخدام كائنات ()Array

المصفوفة هي قائمة مرتبة من القيم، وتُنشَأ عادةً بغرض الدوران على قيم لها مفاتيح (أو فهارس [index]) تبدأ من المفتاح صفر. الذي عليك أن تعرفه هو أنَّ المصفوفات هي مجموعات مرقمة عدديًا، بينما تحتوي الكائنات على خاصيات لها أسماء مرتبطة مع قيم بترتيبٍ غير رقمي. بشكلٍ أساسي، تُستخدَم الأرقام في المصفوفات للبحث عن قيمة، بينما تَستخدِم الكائنات أسماءً مُعرَّفةً من المستخدم للخاصيات. لا تملك JavaScript نظريًا مصفوفاتٍ ترابطية (associative arrays) لكن يمكن أن تُستخدم الكائنات للقيام بنفس وظيفة المصفوفات الترابطية.

سأُخزِّن في المثال الآتي أربع سلاسل نصية في المصفوفة `myArray` والتي سأتمكن من الوصول إليها عبر فهرسٍ رقمي (numeric index). وسأقارن ذلك بكائنٍ يحاكي مصفوفةً ترابطيةً (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = ['blue', 'green', 'orange', 'red'];

// الناتج blue، بعد استخدامنا للفهرس 0 للوصول إلى
// السلسلة النصية في مصفوفة myArray
console.log(myArray[0]);

// مصفوفةٌ ترابطيةٌ، وهي في الواقع كائنٌ
var myObject = {
    'blue': 'blue',
```

```
    'green': 'green',
    'orange': 'orange',
    'red': 'red'
};

console.log(myObject['blue']); // الناتج: blue

</script></body></html>
```

ملاحظات	
	- يمكن للمصفوفات أن تحتوي أيَّ نوعٍ من القيم، ويمكن تحديث أو حذف تلك القيم في أيّ وقت. - المصفوفات من النوع `Array()` هي نوعٌ خاصٌ من الكائن `Object()`. أي أنَّ النسخ المُنشأة من `Array()` هي في الواقع كائنات `Object()` مُضافٌ إليها بعض الدوال الإضافية (مثلًا `length`. بالإضافة إلى الفهرس الرقمي المُضمَّن فيها). - يُشار عادةً للقيم الموجودة في مصفوفة «بالعناصر» (elements).

2. معاملات الدالة البانية Array()

يمكنك أن تُمرِّر قيم نسخة المصفوفة التي تريد إنشاءها إلى الدالة البانية كمعاملات مفصولٌ بينها بفاصلة (مثلًا `new Array('foo', 'bar');`). يمكن أن تستقبل الدالة البانية `Array()` ما مقداره 4,294,967,295 وسيطًا.

لكن إن مُرِّر معاملٌ وحيدٌ إلى الدالة البانية `Array()` وكانت قيمة هذا المعامل رقميةً (مثلًا 1 أو 123 أو 1.0) فستُستخدَم قيمة ذاك المعامل لإعداد الخاصية `length` في المصفوفة، ولن تُستخدَم كقيمة موجودة داخل المصفوفة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var foo = new Array(1, 2, 3);
var bar = new Array(100);

console.log(foo[0], foo[2]); // الناتج: '1 3'
console.log(bar[0], bar.length); // الناتج: 'undefined 100'

</script></body></html>
```

3. الخاصيات والدوال الموجودة في ()Array

يملك الكائن `Array()` الخاصيات الآتية (باستثناء الخاصيات والدوال التي يرثها):

- الخاصيات (مثلًا `Array.prototype`):
  - `prototype`

4. الخاصيات والدوال الموجودة في الكائنات من نوع ()Array

تملك الكائنات ذات النوع `Array()` الخاصيات والدوال الآتية (باستثناء الخاصيات والدوال التي ترثها):

- الخاصيات (مثلًا `var myArray = ['foo', 'bar']; myArray.length;`):
  - `constructor`
  - `index`
  - `input`
  - `length`
- الدوال (مثلًا `var myArray = ['foo']; myArray.pop();`):
  - `pop()`
  - `push()`
  - `reverse()`
  - `shift()`
  - `sort()`
  - `splice()`
  - `unshift()`
  - `concat()`
  - `join()`
  - `slice()`

5. إنشاء المصفوفات

كأغلبيـة الكائنـات في JavaScript، يمكن إنشـاء كـائن مصـفوفة جديـد باسـتخدام المعامل `new` بالإضافة إلى الدالة البانية `Array()`، أو عبر استعمال الشكل المختصر.

سأُنشِئ في المثال الآتي المصفوفة `myArray1` مع قيم مُعرَّفة مسبقًا باستخدام الدالة البانيـة

()Array، ومن ثم سأُنشِئ المصفوفة myArray2 عبر الشكل المختصر (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// Array() إنشاء مصفوفة عبر الدالة البانية
var myArray1 = new Array('blue', 'green', 'orange', 'red');

// الناتج: ["blue", "green", "orange", "red"]
console.log(myArray1);

// إنشاء مصفوفة عبر الشكل المختصر
var myArray2 = ['blue', 'green', 'orange', 'red'];

// الناتج: ["blue", "green", "orange", "red"]
console.log(myArray2);

</script></body></html>
```

من الشائع تعريف الدالة عبر الشكل المختصر، لكن علينا أن نعرف أنَّ هذا الشكل المختصر ما هو إلا إخفاءٌ لاستعمال الدالة البانية ()Array.

ملاحظات	- عمليًا، كل ما ستحتاج إليه هو الشكل المختصر. - بغض النظر عن طريقة إنشاء المصفوفة، فإن لم توفِّر إليها أيّة قيمة مُعرَّفة مسبقًا فستُنشَأ ولكنها ببساطة لن تحتوي أيّة قيم.

6. إضافة وتحديث القيم في المصفوفات

يمكن إضافة قيمةٍ ما إلى مصفوفة في أي فهرس تريد. سنُضيف في المثال الآتي قيمةً ذاتُ الفهرس الرقمي 50 إلى مصفوفةٍ فارغة. لكن ماذا عن الفهارس التي تسبق الرقم 50؟ حسنًا، كما أخبرتك منذ قليل أنَّك تستطيع إضافة قيمة إلى مصفوفة في أيّ فهرس وفي أيّ وقت. لكن إن أضفتَ قيمةً إلى الفهرس الرقمي 50 في مصفوفةٍ فارغة، فستملَأ JavaScript جميع الفهارس اللازمة قبل ذاك الفهرس بقيم undefined (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = [];
myArray[50] = 'blue';

// الناتج 51 (سيبدأ إحصاء القيم من الصفر)
// لأنّ JS أنشأت القيم من 0 إلى 50 قبل القيمة blue
console.log(myArray.length);

</script></body></html>
```

إضافةً إلى ذلك –وبأخذ الطبيعة الديناميكية للغة JavaScript وأنَّ لغة JavaScript لغةٌ متساهلةٌ في أنواع البيانات (not strongly typed)– فيمكن أن تُحدَّث قيمةٌ في مصفوفةٍ في أيّ وقتٍ ويمكن أن تكون القيمةُ الموجودةُ في فهرسٍ ما أيّةَ قيمةٍ مسموحةٍ. سأغيّرُ في المثال الآتي القيمةَ الموجودةَ في الفهرس الرقمي 50 إلى كائنٍ (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = [];
myArray[50] = 'blue';
// Object() تغيير نوع الكائن من سلسلة نصيّة إلى كائن من نوع
myArray[50] = {'color': 'blue'};

console.log(myArray[50]); // الناتج: 'Object {color="blue"}'

// عبر استخدمنا للأقواس المربعة، سنتمكن من الوصول إلى
// فهرس في المصفوفة، ثم إلى الخاصية color
console.log(myArray[50]['color']); // الناتج: 'blue'

// استخدام النقطة للوصول إلى الخاصية
console.log(myArray[50].color); // الناتج: 'blue'

</script></body></html>
```

7. الفهارس وطول المصفوفة

تبدأ أيّة مصفوفة ترقيمَ عناصرِها بدءًا من الصفر، وهذا يعني أنَّ أوّل خانةٍ يمكن أن تُخزَّن البيانات فيها في المصفوفة تشابه `myArrray[0]`. وقد يربكك هذا قليلًا إذا أنشأتُ مصفوفةً فيها قيمةٌ وحيدةٌ، فإن فهرس القيمة هو `0` لكن «طول» (length) المصفوفة هو `1`. عليك أن تفهم أنَّ طول المصفوفة يُمثِّل عدد القيم الموجودة فيها، بينما الفهرس الرقمي في المصفوفة يبدأ من الرقم `0`.

ستحتوى القيمة blue -في المثال الآتي- في المصفوفة myArray في الفهرس الرقمي 0، لكن لمّا كانت المصفوفة تحتوي على قيمةٍ وحيدةٍ، فإن «طولها» هو 1 (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// الفهرس 0 يحتوي على القيمة النصية 'blue'
var myArray = ['blue'];
console.log(myArray[0]); // الناتج: 'blue'
console.log(myArray.length); // الناتج: 1

</script></body></html>
```

8. إنشاء مصفوفات ذات خاصية length مُعرفَّة مسبقًا

كما ذكرتُ سابقًا، عند تمرير معامل رقمي وحيد إلى الدالة ()Array، فيمكننا تعريف طول المصفوفة مسبقًا (أي عدد القيم التي ستحتويها). وفي هذه الحالة ستُصدِرُ الدالةُ البانيةُ استثناءً وتفترض أنَّك تريد ضبط طول المصفوفة ولا تريد أن تملأ القيم في المصفوفة عند إنشائها.

سنُهيّئ -في مثالنا الآتي- المصفوفة myArray بطولٍ قدره 3. أكرِّر أننا نضبط طول المصفوفة، ولا نُمرِّر إليها قيمةً لتُخزَّن في الفهرس 0 (length) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = new Array(3);
// الناتج 3، لأننا مرّرنا معاملًا رقميًا وحيدًا
```

```
console.log(myArray.length);
console.log(myArray[0]); // الناتج: undefined

</script></body></html>
```

ملاحظات

- سيؤدي توفير طول مُعرَّف مسبقًا للمصفوفة إلى إعطاء كل فهرس رقمي (بدءًا من الصفر حتى الطول المُحدَّد) قيمةً مرتبطةً به هي undefined.

- ربما تتساءل بينك فيما إذا كان من الممكن إنشاء مصفوفة تحتوي على قيمة رقمية وحيد: نعم يمكنك ذلك باستخدام الشكل المختصر (أي var myArray = [4]).

9. ضبط خاصية length قد يؤدي إلى إضافة أو حذف القيم

خاصية length الموجودة لكائن مصفوفة يمكن أن تُستخدَم للحصول على «طول» المصفوفة أو ضبطه. وكما وضحنا أعلاه، فإن ضبط الطول إلى رقم أكبر من عدد القيم الموجودة في المصفوفة سيؤدي إلى إضافة قيم undefined إليها. لكن ما قد لا تتوقعه هو أنَّك تستطيع إزالة القيم من المصفوفة عبر ضبط قيمة الطول إلى عدد أقل من عدد القيم الموجودة في المصفوفة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = ['blue', 'green', 'orange', 'red'];
```

```
console.log(myArray.length); // الناتج: 4
myArray.length = 99;
// الناتج 99، تذكر أننا ضبطنا الطول، وليس الفهرس //
console.log(myArray.length);

// حذفنا جميع القيم ما عدا واحدة //
// لذا ستُحذَف القيمة المرتبطة بالفهرس 1 //
myArray.length = 1;
console.log(myArray[1]); // الناتج: undefined

console.log(myArray); // الناتج: '["blue"]'

</script></body></html>
```

10. المصفوفات التي تحتوي مصفوفاتٍ أخرى (أي المصفوفات متعددة الأبعاد)

لمّا كان بإمكان المصفوفة أن تخُزِّن أيّة قيمة في JavaScript، فيمكن للمصفوفات أن تحتوي على مصفوفات أخرى؛ وعندما نفعل ذلك فستُسمى المصفوفة التي تحتوي المصفوفات الأخرى «بالمصفوفة متعددة الأبعاد» (multidimensional array). يمكننا الوصول إلى المصفوفات الداخلية عبر الأقواس المربعة. ستُنشِئ في المثال الآتي مصفوفةً (بالشكل المختصر) التي تحتوي على مصفوفةٍ، وداخلها ستُنْشِئ مصفوفةً أخرى، والتي تحتوي بدورها على مصفوفةٍ التي تحتوي على قيمةٍ نصيةٍ في الفهرس `0` (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = [[[['4th dimension']]]];
console.log(myArray[0][0][0][0]); // الناتج: '4th dimension'

</script></body></html>
```

صحيحٌ أنَّ الشيفرة السابقة «سخيفة»، لكنك تستطيع أن تفهم منها أنَّ المصفوفات يمكن أن تحتوي على مصفوفاتٍ أخرى ويمكنك أن تستعمل أي عدد تريده من المصفوفات المتشعبة (أو الأبعاد).

11. الدوران على عناصر المصفوفة أماميًا وخلفيًا

أبسط وأسرع طريقة للدوران على عناصر مصفوفة هو استخدام حلقة `while`.

سأريك في المثال الآتي كيفية الدوران من بداية المصفوفة إلى نهايتها (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = ['blue', 'green', 'orange', 'red'];

// تخزين طول المصفوفة
// لكي نتجنب المحاولات غير الضرورية للوصول إلى الخاصية
var myArrayLength = myArray.length;
var counter = 0; // ضبط العدّاد
```

```
// تنفيذ ما يلي إذا كان العدّاد أصغر من طول المصفوفة
while (counter < myArrayLength) {
    // الناتج: 'blue', 'green', 'orange', 'red'
    console.log(myArray[counter]);
    counter++; // إضافة 1 إلى العدّاد
}

</script></body></html>
```

ويمكننا أيضًا المرور من نهاية المصفوفة إلى بدايتها (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myArray = ['blue', 'green', 'orange', 'red'];

var myArrayLength = myArray.length;
// إذا لم يكن الطول مساويًا للصفر
// فنفِّذ محتوى الحلقة وأنقص منه واحدًا
while (myArrayLength--) {
    // الناتج: 'red', 'orange', 'green', 'blue'
    console.log(myArray[myArrayLength]);
}

</script></body></html>
```

إذا كنتَ تتساءل لماذا لم أستخدم حلقات for هنا، فالسبب هو أنَّ حلقات while أبسط

وفيها أجزاءٌ «متحركةٌ» أقل وأرى شخصيًا أنَّ قراءتها أسهل.

الفصل العاشر:

السلاسل النصية ()String

10

1. لمحة نظرية عن الكائن ()String

تُستخدَم الدالة البانية `String()` لإنشاء كائنات نصيّة ولإعادة القيم النصيّة الأولية.

سأوضِّح في الشيفرة التالية كيفية إنشاء السلاسل النصية في JavaScript (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// إنشاء كائن باستخدام الكلمة المحجوزة new
// والدالة البانية String()
var stringObject = new String('foo');

// الناتج: foo {0 = 'f', 1 = 'o', 2 = 'o'}
console.log(stringObject);
console.log(typeof stringObject); // الناتج: 'object'

// إنشاء سلسلة نصيّة أوليّة من
// الدالة البانية String() مباشرةً
// لاحظ عدم استخدام الكلمة المحجوزة new
var stringObjectWithOutNewKeyword = String('foo');
console.log(stringObjectWithOutNewKeyword); // الناتج: 'foo'
// الناتج: 'string'
console.log(typeof stringObjectWithOutNewKeyword);

// إنشاء سلسلة نصيّة أوليّة
// (ستُستخدَم الدالة البانية وراء الكواليس)
```

```
var stringLiteral = 'foo';
console.log(stringLiteral); // الناتج: foo
console.log(typeof stringLiteral); // الناتج: 'string'

</script></body></html>
```

2. معاملات الدالة البانية ()String

تقبل الدالة البانية `String()` معاملًا وحيدًا هو السلسلة النصية المُراد إنشاؤها. سأُنشِئ في المثال الآتي المتغير `stringObject` الذي يحتوي على السلسلة النصية «foo» (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// إنشاء كائن سلسلة نصيّة
var stringObject = new String('foo');

// الناتج: 'foo {0="f", 1="o", 2="o"}'
console.log(stringObject);

</script></body></html>
```

ملاحظة	الكائنات المُنشَأة من الدالة البانية `String()` عندما تُستخدَم الكلمة المحجوزة `new` معها هي كائناتٌ معقدةٌ. ويجب أن تتفادى فعل ذلك (وتستخدم السلاسل النصية الأوليّة بدلًا من ذلك) بسبب المشاكل المحتمل

وقوعك فيها مع المعامل typeof؛ إذ سيُعيد المعامل typeof القيمة object لكائنات السلاسل النصية بدلًا من القيمة string التي تتوقعها. إضافةً إلى أنَّه من الأسرع والأسهل استخدام السلاسل النصية الأولية.

3. الخاصيات والدوال الموجودة في ()String

يملك الكائن ()String الخاصيات الآتية (باستثناء الخاصيات والدوال التي يرثها):

- الخاصيات (مثلًا String.prototype):
  - prototype
- الدوال (مثلًا ;()String.formCharCode):
  - ()fromCharCode

4. الخاصيات والدوال الموجودة في الكائنات من نوع ()String

تملك الكائنات ذات النوع ()String الخاصيات والدوال الآتية (باستثناء الخاصيات والدوال التي ترثها):

- الخاصيات (مثلًا ;var myString = 'foo'; myString.length):
  - constructor
  - length
- الدوال (مثلًا ;()var myString = 'foo'; myString.toLowerCase):
  - ()charAt
  - ()charCodeAt

- `concat()`
- `indexOf()`
- `lastIndexOf()`
- `localeCompare()`
- `match()`
- `replace()`
- `search()`
- `slice()`
- `split()`
- `substr()`
- `substring()`
- `toLocaleLowerCase()`
- `toLocaleUpperCase()`
- `toLowerCase()`
- `toString()`
- `toUpperCase()`
- `valueOf()`

الفصل الحادي عشر:

الأعداد Number()

11

1. لمحة نظرية عن الكائن ()Number

تُستخدَم الدالة البانية `Number()` لإنشاء كائنات عددية ولإنشاء القيم العددية الأوليّة.

سأوضِّح طريقة إنشاء القيم العددية في JavaScript في المثال الآتي (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// إنشاء كائن عددي باستخدام الكلمة المحجوزة new
// والدالة البانية Number()
var numberObject = new Number(1);
console.log(numberObject); // الناتج: 1
console.log(typeof numberObject); // الناتج: 'object'

// إنشاء قيمة عددية أوليّة باستخدام
// الدالة البانية Number() دون new
var numberObjectWithOutNew = Number(1);
// الناتج: 1
console.log(numberObjectWithOutNew);
// الناتج: 'number'
console.log(typeof numberObjectWithOutNew);

// إنشاء قيمة عددية أوليّة
// (ستُستخدَم الدالة البانية وراء الكواليس)
var numberLiteral = 1;
console.log(numberLiteral); // الناتج: 1
```

```
console.log(typeof numberLiteral); // الناتج: 'number'

</script></body></html>
```

2. الأعداد الصحيحة والأعداد العشرية

إما أن تكون الأعداد في JavaScript أعدادًا صحيحةً (integers) أو أعدادًا عشرية (يسمونها أيضًا «أعدادٌ ذات فاصلةٍ عائمة» [floating point]). سأنُشِئ في الشيفرة الآتية عددًا صحيحًا أوليًا وعددًا عشريًا أوليًا. وهذا هو أكثر استعمالٍ شائعٍ للأعدادِ في JavaScript (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var integer = 1232134;
console.log(integer); // الناتج: '1232134'

var floatingPoint = 2.132;
console.log(floatingPoint); // الناتج: '2.132'

</script></body></html>
```

ملاحظة	يمكن التعبير عن الأعداد أيضًا بالنظام الست عشري (hexadecimal) أو النظام الثماني (octal) في JavaScript، لكن ذلك نادرٌ ولا نفعله عادةً.

3. معاملات الدالة البانية Number()

تأخذ الدالة البانية `Number()` معاملًا وحيدًا هو القيمة العددية التي ستُنشِئها. ستُنشِئ في المثال الآتي كائنًا عدديًا باسم `numberOne` يحتوي على القيمة 456 (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var numberOne = new Number(456);

console.log(numberOne); //الناتج: '456{}'

</script></body></html>
```

ملاحظة	النسخ المُنشأة من الدالة البانية `Number()` عندما نستخدمها مع الكلمة المحجوزة `new` هي كائناتٌ معقدةٌ. يجب أن تتجنب إنشاء قيم عددية باستخدام الدالة البانية `Number()` (استخدم القيم الأوليّة بدلًا منها) بسبب المشاكل المحتمل وقوعك فيها مع المعامل `typeof`؛ إذ سيُعيد المعامل `typeof` القيمة للكائنات العددية بدلًا من القيمة `number` التي تتوقعها. إضافةً إلى أنّه من الأسرع والأسهل استخدام القيم العددية الأولية.

4. الخاصيات والدوال الموجودة في Number()

يملك الكائن `Number()` الخاصيات الآتية:

- الخاصيات (مثلًا `Number.prototype`):

- `MAX_VALUE`
- `MIN_VALUE`
- `NaN`
- `NEGATIVE_INFINITY`
- `POSITIVE_INFINITY`
- `prototype`

5. الخاصيات والدوال الموجودة في الكائنات من نوع ()Number

تملك الكائنات ذات النوع `Number()` الخاصيات والدوال الآتية (باستثناء الخاصيات والدوال التي ترثها):

- الخاصيات (مثلًا `var myNumber = 5; myString.constructor;`):
  - `constructor`
- الدوال (مثلًا `var myNumber = 1.00324; myNumber.toFixed();`):
  - `toExponential()`
  - `toFixed()`
  - `toLocaleString()`
  - `toPrecision()`
  - `toString()`
  - `valueOf()`

الفصل الثاني عشر:

القيم المنطقية Boolean()

12

1. لمحة نظرية عن الكائن ()Boolean

يمكن أن تُستخدم الدالة البانية `Boolean()` لإنشاء كائنات منطقية، بالإضافة إلى القيم المنطقية الأولية، والتي تُمثِّل `true` أو `false`.

سأوضِّح إنشاء القيم المنطقية في JavaScript في المثال الآتي (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// إنشاء كائن منطقي باستخدام الكلمة المحجوزة new
// والدالة البانية Boolean()
var myBoolean1 = new Boolean(false);
console.log(typeof myBoolean1); // الناتج: 'object'

// إنشاء قيمة منطقية أوليّة باستدعاء
// الدالة البانية Number() دون استخدام new
var myBoolean2 = Boolean(0);
console.log(typeof myBoolean2); // الناتج: 'boolean'

// إنشاء قيمة منطقية أوليّة
// (ستُستدعى الدالة البانية وراء الكواليس)
var myBoolean3 = false;
// الناتج: 'boolean'
console.log(typeof myBoolean3);
// الناتج: false false false
console.log(myBoolean1, myBoolean2, myBoolean3);
```

```
</script></body></html>
```

2. معاملات الدالة البانية ()Boolean

الدالة البانية `Boolean()` تأخذ معاملًا وحيدًا الذي سيُحوَّل إلى قيمة منطقية (أي `true` أو `false`). أيّة قيمة في JavaScript ليست `0` أو `-0` أو `null` أو `false` أو `NaN` أو `undefined` أو سلسلة نصية فارغة (`""`) ستُحوَّل إلى `true`. سنُنشِئ في المثال الآتي كائنين منطقيين أحدهما `true` والآخر `false` (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// false = 0 هو Boolean() الوسيط المُمرّر إلى
// foo = false وبالتالي
var foo = new Boolean(0)
console.log(foo);

// true = Math هو Boolean() الوسيط المُمرّر إلى
// bar = true وبالتالي
var bar = new Boolean(Math)
console.log(bar);

</script></body></html>
```

ملاحظة	النسخ المُنشأة من الدالة البانية `Boolean()` عندما نستخدمها مع الكلمة المحجوزة `new` هي كائناتٌ معقدةٌ. يجب أن تتجنب إنشاء قيم منطقية باستخدام الدالة البانية `Boolean()` (استخدم القيم الأوليّة بدلًا منها) بسبب المشاكل المحتمل وقوعك فيها مع المعامل `typeof`؛ إذ سيُعيد المعامل `typeof` القيمة للكائنات العددية بدلًا من القيمة `boolean` التي تتوقعها. إضافةً إلى أنَّه من الأسرع والأسهل استخدام القيم المنطقية الأولية.

3. الخاصيات والدوال الموجودة في ()Boolean

يملك الكائن `Boolean()` الخاصيات الآتية:

- الخاصيات (مثلًا `Boolean.prototype`):
  - `prototype`

4. الخاصيات والدوال الموجودة في الكائنات من نوع ()Boolean

تملك الكائنات ذات النوع `Boolean()` الخاصيات والدوال الآتية (باستثناء الخاصيات والدوال التي ترثها):

- الخاصيات (مثلًا `var myBoolean = false; myBoolean.constructor;`):
  - `constructor`
- الدوال (مثلًا `var myBoolean = false ; myBoolean.toString();`):
  - `toSource()`
  - `toString()`
  - `valueOf()`

5. الكائنات المنطقية غير الأولية ذات القيمة false ستتحول إلى true

الكائنات المنطقية غير الأولية ذات القيمة false المُنشأة من الدالة البانية ()Boolean هي عبارة عن كائنٍ، وستتحول قيمة الكائن إلى true، وبالتالي فعند إنشاء كائن منطقية ذو القيمة false عبر الدالة البانية ()Boolean فإن القيمة نفسها ستتحول إلى true. سأوضِّح في المثال الآتي كيف ستُعتبَر قيمة كائن منطقي true على الرغم من إسناد القيمة false له (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var falseValue = new Boolean(false);

// false لدينا كائن منطقي ذو القيمة
// true لكن تُعتبَر الكائنات دومًا على أنها
console.log(falseValue);

/*
جميع الكائنات –بما فيها الكائنات المنطقية ذات القيمة
true ستُعتبَر –false
*/
if (falseValue) {
    console.log('falseValue is truthy');
}

</script></body></html>
```

إذا احتجتَ لتحويل قيمة غير منطقية إلى قيمةٍ منطقية، فاستخدم الدالة البانية `Boolean()` دون الكلمة المحجوزة `new` وستُعاد قيمةٌ منطقيةٌ أوليةٌ بدلًا من كائنٍ من النوع `Boolean`.

6. قيم بعض الأشياء false والبقية true

كما ذكرتُ سابقًا (لكن الأمر يستحق الذكر مرةً أخرى لأن الأمر متعلقٌ بتحويل مختلف القيم إلى قيم منطقية) إذا كانت القيمة `0` أو `-0` أو `null` أو `false` أو `NaN` أو `undefined` أو سلسلةً نصيةً فارغةً (`""`) فستُحوَّل إلى `false`؛ وأيّة قيمة أخرى في JavaScript ما عدا القيمة سابقة الذكر ستُحوَّل إلى `true` إذا أُستخدِمَت في تعبيرٍ يَستعمِلُ القيمةَ المنطقيةَ (مثلًا `(if (true) {};`) (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// false جميع القيم التالية ستُعتبَر
console.log(Boolean(0));
console.log(Boolean(-0));
console.log(Boolean(null));
console.log(Boolean(false));
console.log(Boolean(''));
console.log(Boolean(undefined));
console.log(Boolean(null));

// true جميع القيم التالية ستُعتبَر
console.log(Boolean(1789));
```

```
// false السلسلة النصية 'false' لا تُمثِّل القيمة المنطقية
console.log(Boolean('false'));
console.log(Boolean(Math));
console.log(Boolean(Array()));

</script></body></html>
```

من الضروري أن تعرف ما هي القيم التي ستحولها JavaScript إلى `false`، وأن تعلم أنَّ ما بقي من القيم سيتحول إلى `true`.

الفصل الثالث عشر:

التعامل مع السلاسل النصية والأعداد والقيم المنطقية الأولية

13

1. ستتحول القيم الأولية إلى كائنات عندما نحاول الوصول إلى خاصياتها

لا يربكنّك أنَّ القيم الأولية للسلاسل النصية والأعداد والقيم المنطقية يمكن أن تُعامل ككائن ذي خاصيات (مثلًا `ture.toString()`)، فعندما نعامِل تلك القيم الأولية ككائنات بمحاولتنا الوصول إلى خاصياتها، فستُنشِئ JavaScript كائنًا من الدالة البانية التابعة للقيمة الأولية، لكي نتمكن من الوصول إلى الخاصيات أو الدوال المرتبطة بذاك العنصر. وبعد انتهائنا من الوصول إلى الخاصيات، فسيُحذَف ذاك الكائن.

هذه الطريقة تسمح لنا بكتابة شيفرات حيث يبدو فيها كما لو أنَّ القيم الأولية هي كائنات. والحقيقة هي أننا عندما نعامِل القيم الأولية ككائن في الشيفرة، فستحولها JavaScript إلى كائنٍ لكي نتمكن من الوصول إلى الخاصيات ومن ثم ستُرجِع JavaScript القيمةَ إلى قيمةٍ أولية. أهم ما في الأمر أن تفهم ما الذي يحدث، وأنَّ JavaScript تفعل ذلك لك خلف الكواليس.

مثالٌ عن السلاسل النصية (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// الكائن من النوع String سيُعامَل ككائن!
var stringObject = new String('foo');
console.log(stringObject.length); // الناتج: 3
console.log(stringObject['length']); // الناتج: 3

// ستحوّل السلسلة النصية الأوليّة إلى كائن
```

```
// عندما نعاملها ككائن

var stringLiteral = 'foo';
console.log(stringLiteral.length); // الناتج: 3
console.log(stringLiteral['length']); // الناتج: 3
console.log('bar'.length); // الناتج: 3
console.log('bar'['length']); // الناتج: 3

</script></body></html>
```

مثالٌ عن الأعداد (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// الكائن من النوع Number سيُعامَل ككائن!
var numberObject = new Number(1.10023);
console.log(numberObject.toFixed()); // الناتج: 1
console.log(numberObject['toFixed']()); // الناتج: 1

// ستحوّل القيمة العددية الأوليّة إلى كائن
// عندما نعاملها ككائن
var numberLiteral = 1.10023;
console.log(numberLiteral.toFixed()); // الناتج: 1
console.log(numberLiteral['toFixed']()); // الناتج: 1
console.log((1234).toString()); // الناتج: '1234'
console.log(1234['toString']()); // الناتج: '1234'
```

```
</script></body></html>
```

مثالٌ عن القيم المنطقية (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// الكائن من النوع Boolean سيُعامَل ككائن!
var booleanObject = new Boolean(0);
console.log(booleanObject.toString()); // الناتج: 'false'
console.log(booleanObject['toString']()); // الناتج: 'false'

// ستحوّل القيمة المنطقية الأوليّة إلى كائن
// عندما نعاملها ككائن
var booleanLiteral = false;
console.log(booleanLiteral.toString()); // الناتج: 'false'
console.log(booleanLiteral['toString']()); // الناتج: 'false'
console.log((true).toString()); // الناتج: 'true'
console.log(true['toString']()); // الناتج: 'true'

</script></body></html>
```

ملاحظة	عندما نحاول الوصول إلى خاصية لقيمة عددية أوليّة مباشرة (أي أنها غير مخزنةٍ في متغير)، فعلينا أولًا «تحديد قيمة العدد» قبل معاملته ككائن (مثلًا `(1).toString()` أو `1..toString()`)، لكن لماذا وضعنا نقطتين؟ تُعتبَر

أول نقطة على أنها الفاصلة العشرية، وليس المعامل الذي يُستخدَم للوصول إلى خاصيات كائن.

2. عليك عادةً استخدام القيم النصية والعددية والمنطقية الأوليّة

القيم الأولية التي تُمثِّل السلاسل النصية أو الأعداد أو القيم المنطقية هي أسرع في الكتابة وشكلها مختصر.

يجب عليك في غالبية الأوقات استخدام القيم الأوليّة، فبالإضافةِ إلى ما سبق: تعتمد دقة ناتج المعامل typeof على كيفية إنشائك للقيمة (قيم أوليّة أو كائنات). فلو أنشأت كائنَ سلسلةٍ نصيةٍ أو عددٍ أو قيمةٍ منطقيةٍ فسيقول المعامل typeof أنَّ نوع تلك القيمة هو object. أما لو استخدمتَ القيم الأولية فسيعيد المعامل typeof سلسلة نصية تحتوي على النوع الحقيقي للقيمة (مثلًا في typeof 'foo' يكون الناتج 'string').

سأوضِّح ما سبق في الشيفرة الآتية (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// كائنات لسلسلةٍ نصيةٍ ولعددٍ ولقيمةٍ منطقيةٍ
console.log(typeof new String('foo')); // الناتج: 'object'
console.log(typeof new Number(1)); // الناتج: 'object'
console.log(typeof new Boolean(true)); // الناتج: 'object'

// قيم أوليّة لسلسلةٍ نصيةٍ ولعددٍ ولقيمةٍ منطقيةٍ
```

```
console.log(typeof 'foo'); // الناتج: 'string'
console.log(typeof 1); // الناتج: 'number'
console.log(typeof true); // الناتج: 'boolean'

</script></body></html>
```

إذا كان برنامجك يعتمد على المعامل `typeof` للتعرف على القيم الأولية السلاسل النصية أو الأعداد أو القيم المنطقية، فعليك أن تتفادى حينها استخدام الدوال البانية `String()` و `Number()` و `Boolean()`.

الفصل الرابع عشر:

القيمة null

14

1. لمحة نظرية عن استخدام القيمة null

يمكنك استخدام القيمة null لكي تُصرِّح أن إحدى خاصيات الكائن لا تملك قيمةً. عمومًا إذا ضُبِطَت خاصيةٌ لتحتوي على قيمةٍ ما، لكن تلك القيمة غير متاحةٍ لسببٍ من الأسباب، فيجب أن تُستعمَل القيمة null للإشارة إلى أنَّ قيمة تلك الخاصية فارغة (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// الخاصية foo تنتظر إسناد قيمةٍ لها
// لذا سنضبط قيمتها الابتدائية إلى null
var myObjectObject = {foo: null};

console.log(myObjectObject.foo); // الناتج: 'null'

</script></body></html>
```

ملاحظة	لا تخلط بين null و undefined. تُستعمَل undefined من JavaScript لإخبارك أنَّ شيئًا ما ناقصٌ أما null فهي موجودةٌ للسماح لك بالإشارة إلى أنَّ الخاصية تتوقع إسناد قيمة إليها لكن تلك القيمة غير متوافرة بعد.

2. المعامل typeof سيُعيد object لقيم null

إذا استخدمتَ المعامل typeof على قيمة null فسيُعيد object. وإذا احتجتَ إلى التحقق من أنَّ القيمة هي null فالحل المثالي هو النظر إن كانت القيمة التي تريد التحقق منها

مساويةً إلى null. سنستخدم في المثال الآتي المعامل === للتأكد من أننا نتعامل مع قيمة null (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject = null;

// الناتج object، ولن تستفيد حقيقةً من هذه النتيجة
console.log(typeof myObject);
// الناتج true، فالمساواة محققة لقيم null الفعلية
console.log(myObject === null);

</script></body></html>
```

ملاحظة

عندما تتحقق من قيمة null، فاحرص على استخدام === لأنَّ المعامل == لا يُفرِّق بين null و undefined.

الفصل الخامس عشر:

القيمة undefined

15

1. لمحة نظرية عن القيمة undefined

تُستعمل القيمة `undefined` في JavaScript لغرضين مختلفين.

الهدف من أول غرض هو الإشارة إلى أنَّ متغيرًا معرَّفًا (مثلًا `var foo`) لم تُسنَد إليه قيمة.
أما الغرض الثاني فهو للإشارة إلى أنَّ الخاصية التابعة لكائن والتي تحاول الوصول إليها غير معرفة (أي لا توجد خاصية بهذا الاسم) وليست معرَّفةً أيضًا في سلسلة prototype.

سأريك طريقتَي استخدام القيمة `undefined` في JavaScript (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// التصريح عن متغير
var initializedVariable;

console.log(initializedVariable); // الناتج: undefined
// التأكّد أنّ JavaScript أعادت undefined
console.log(typeof initializedVariable);

var foo = {};

// الناتج undefined، فلا توجد الخاصية bar في الكائن foo
console.log(foo.bar);
// التأكّد أنّ JavaScript أعادت القيمة undefined
console.log(typeof foo.bar);
```

```
</script></body></html>
```

ملاحظة

من المستحسن أن نسمح للغة JavaScript أن تستخدم undefined فقط. يجب ألّا تسمح لنفسك بأن تضبط قيمة أحد المتغيرات إلى undefined (أي foo = undefined)، وعليك بدلًا من ذلك استخدام null إن كنتَ تود الإشارة إلى أنّ قيمة المتغير أو الخاصية غير متوافرة في اللحظة الحالية.

2. نسخة JavaScript ECMA-262 الإصدار الثالث (وما بعده) تُعرِّف المتغير undefined في المجال العام

على عكس الإصدارات السابقة، الإصدار الثالث من JavaScript ECMA-262 يملك متغيرًا عامًا اسمه undefined موجودٌ في المجال العام. ولأنَّ المتغير مصرَّحٌ عنه ولم تُسنَد إليه قيمة، فإن قيمة المتغير undefined هي undefined (مثال حي):

```
<!DOCTYPE html><html lang="en"><body><script>

// التأكد من أنّ undefined هو متغيرٌ في المجال العام
console.log(undefined in this); // الناتج: true

</script></body></html>
```

الفصل السادس عشر:

الدوال الرياضيّة

16

1. لمحة نظرية عن الكائن Math

يحتوي الكائن Math على خاصيات ودوال ساكنة (static) للتعامل الرياضي مع الأعداد أو توفير ثوابت رياضية (مثلًا Math.PI). هذا الكائن مُضمَّن في لغة JavaScript، ولا يعتمد على الدالة البانية ()Math لإنشاء كائنات.

ملاحظة	قد يبدو غريبًا عليك أن الكائن Math يبدأ بحرفٍ كبير على الرغم من أنَّك لم تُنشِئ نسخةً منه عبر الدالة البانية ()Math. لكن لا ترتبك بذلك. واعلم أنَّ JavaScript تضبط هذا الكائن لك.

2. خاصيات ودوال الكائن Math

يملك الكائن Math الخاصيات والدوال الآتية:

- الخاصيات (مثلًا ;Math.PI):
  - E
  - LN2
  - LN10
  - LOG2E
  - LOG10E
  - PI
  - SQRT1_2
  - SQRT2

- الدوال (مثلًا `Math.random();`):
  - `abs()`
  - `acos()`
  - `asin()`
  - `atan()`
  - `anat.()`
  - `ceil()`
  - `cos()`
  - `exp()`
  - `floor()`
  - `log()`
  - `max()`
  - `min()`
  - `pow()`
  - `random()`
  - `round()`
  - `sin()`
  - `sqrt()`
  - `tan()`

3. Math ليست دالةً بانيةً

الكائن `Math` لا يُشابه غيره من الكائنات المُهيّئة مسبقًا. فالكائن `Math` مُنشَأٌ لاستضافة

الخاصيات والدوال الساكنة، والجاهزة للاستخدام عند التعامل مع الأرقام. تذكَّر أنَّه لا توجد طريقةٌ لإنشاء نسخة من Math، إذ لا توجد دالةٌ بانيةٌ.

4. الكائن Math يملك ثوابت لا تستطيع تغيير قيمتها

الكثير من خاصيات الكائن Math هي ثوابت لا يمكن تغيير قيمتها. ولأن هذا يختلف عن الطبيعة المتغيرة للكائنات في JavaScript، فستُذكَر هذه الخاصيات بأحرفٍ كبيرة (مثلًا Math.PI). لا تخلط بين الخاصيات الثابتة وبين الدوال البانية بسبب كتابة الحرف M بحرفٍ كبير. ببساطة تلك الخاصيات هي خاصياتٌ لا يمكن تغيير قيمتها.

ملاحظة	لا يُسمَح للمستخدم بتعريف ثوابت في نسخة JavaScript 1.5, ECMA-262 الإصدار الثالث. لكن الإصدار السادس من معيار ECMAScript (أي ES6) تضمّن الكلمة المحجوزة const لتعريف الثوابت.

الملحق الأول:

مراجعة



تُلخِّص النقط الآتية ما يجب أن تكون قد تعلمته أثناء قراءتك لهذا الكتاب (وانتباهك للأمثلة التي فيه). اقرأ كل نقطة بتمعّن، وإن لم تفهمها فارجع إلى الفصل الذي يتحدث عنها في الكتاب.

- يتألف الكائن من مجموعة من الخاصيات التي لها أسماءٌ وتُخزَّن فيها قيمٌ
- يمكن لكل شيء تقريبًا في JavaScript أن يسلك سلوك كائن؛ فالقيم المعقدة هي كائنات، والقيم الأوليّة يمكن معاملتها ككائنات، وهذا هو السبب وراء سماعك للناس يقولون أنَّ كل شيءٍ في JavaScript هو كائن.
- تُنشَأ الكائنات باستدعاء الدالة البانية عبر الكلمة المحجوزة `new`، أو عبر استخدام الشكل المختصر لإنشاء أنواع معيّنة من الكائنات.
- الدوال البانية هي كائنات (أي أنَّها كائنات `Function()`)، وبالتالي يمكن إنشاء كائنات داخل كائنات في JavaScript.
- تــوفِّر JavaScript تســع دوال بانيــة هي: `Object()` و `Array()` و `String()` و `Number()` و `Boolean()` و `Function()` و `Date()` و `RegExp()` و `Error()`. الدوال البانية `String()` و `Number()` و `Boolean()` لها وظيفتان، فهي تُعيد القيم الأولية وتوفِّر كائنات لتلك القيم عند الحاجة؛ لذا سنتمكّن من معاملة القيم الأولية ككائنات.
- القيم `null` و `undefined` و `"string"` و `10` و `true` و `false` هي قيمٌ أوليّة، وليس لها طبيعة «كائنية» إلا إذا عاملتها ككائن.
- عندما تُستدعى الدوال البانية `Object()` و `Array()` و `String()` و `Number()`

و ‎`Boolean()`‎ و ‎`Function()`‎ و ‎`Date()`‎ و ‎`RegExp()`‎ و ‎`Error()`‎ باستخدام الكلمة المحجوزة `new`، فسيُنشأ كائنٌ يسمى «الكائن المعقد» (complex object).

- القيم الأولية `"string"` و `10` و `true` و `false` لا تملك أيّة خصائص من خصائص الكائنات إلى أن تُعامل ككائنات، وحينئذٍ ستُنشِئ JavaScript وراء الكواليس كائنًا مؤقتًا يجعل من الممكن معاملة تلك القيم ككائنات.
- تُخزَّن القيم الأولية بقيمتِها، وعندما تُنسَخ فستُنسَخ قيمتُها إلى المتغير الجديد. أما قيم المتغيرات المعقدة فهي تُخزَّن بمرجعيتها، وعندما تُنسَخ فستُنسَخ المرجعية.
- تكون القيم الأولية مساويةً إلى القيم الأولية الأخرى عندما تتساوى بالقيمة، أما الكائنات المعقدة فتتساوى عندما يشير المتغيران إلى نفس الكائن.
- بسبب طبيعة الكائنات المعقدة، فإن كائنات JavaScript تتسم بأنها ديناميكية.
- لغة JavaScript قابلة للتعديل، وهذا يعني أنَّ خاصيات الكائنات المُضمَّنة في أساس اللغة والكائنات المُعرَّفة من المستخدم يمكن تعديلها في أيّ وقتٍ
- يمكن ضبط أو تحديث أو الحصول على خاصيات الكائن عبر استخدام النقط أو عبر استخدام الأقواس. من الأفضل استخدام الأقواس عندما يكون اسم خاصية الكائن على شكل تعبير أو أن يمُثِّل كلمةً محجوزةً (مثلًا ‎`Array['prototype']['join'].apply()`‎).
- عندما نُشير إلى خاصيات الكائن، فستُستخدَم سلسلة prototype للبحث في نسخة الكائن المُستخدَمة، فإن لم يُعثَر عليها هناك، فسيُبحَث عنها في خاصية `prototype` للدالة

البانية للكائن؛ وإن لم يُعثَر عليها هنالك -ولأنَّ قيمة الخاصية `prototype` هي كائنٌ مُنشأٌ من الدالة البانية `Object()`- فسيتم البحث عن الخاصية في الخاصية `prototype` للدالة البانية `Object()` (أي `Object.prototype`)؛ وإن لم يُعثَر على الخاصية هنالك، فستُعتَبَر هذه الخاصية `undefined`.

- تمثِّل سلسلة prototype طريقة الوراثة الموجودة في JavaScript (أي الوراثة من الكائن `prototype`).
- بسبب البحث في سلسلة prototype (أي الوراثة منها)، فإنَّ جميع الكائنات ترث من `Object()` لأنَّ خاصية `prototype` نفسها هي كائن `Object()`.
- الدوال في JavaScript هي كائنات لها خاصيات وقيم.
- الكلمة المحجوزة `this` -عندما تُستخدَم داخل دالةٍ ما- هي طريقةٌ عامةٌ للإشارة إلى الكائن الذي يحتوي الدالة.
- قيمة الكلمة المحجوزة `this` تُحدَّد أثناء التشغيل بناءً على السياق الذي تُستدعى فيه الدالة.
- عند استخدام الكلمة المحجوزة `this` في المجال العام، فستُشير إلى الكائن الرئيسي.
- تستخدم لغة JavaScript الدوال لإنشاء المجالات.
- توفِّر JavaScript المجال العام، وهو المجال الذي تتواجد فيه جميع شيفرات JavaScript.
- تُنشِئ الدوال (خصوصًا الدوال المتشعبة داخل بعضها) سلسلةً من المجالات تُستعمَل للبحث عن قيمة أحد المتغيرات.

- تُحدَّد سلسلة المجال بناءً على طريقة كتابة الشيفرة، وليس بالضرورة اعتمادًا على سياق الاستدعاء. وهذا يسمح للدالة بالوصول إلى المجال الذي عُرِّفَت فيه أوَّل مرة، حتى لو استدعيت تلك الدالة من سياقٍ مختلف. يؤدي ما سبق إلى إنشاء التعابير البرمجية المغلقة (closures).
- التعابير والمتغيرات المُعرَّفة داخل دالة دون استخدام `var` ستصبح خاصياتٍ عامةً. لكن التعليمات الموجودة داخل مجال الدالة ستبقى في مجالها.
- الدوال والمتغيرات المُعرَّفة دون `var` في المجال العام ستصبح خاصياتٍ للكائن الرئيسي.
- الدوال والمتغيرات المعرَّفة باستخدام `var` في المجال العام ستصبح متغيراتٍ عامةً.

الملحق الثاني:

الخلاصة



أرجو بعد قراءتك لهذا الكتاب أن تكون مزودًا بالمعلومات اللازمة لتفهم كيف تعمل مكتبة JavaScript التي تستخدمها، أو أن تكون على درايةٍ كافيةٍ بلغة JavaScript لتكتب الشيفرات الخاصة بك. وفي كلا الحالتين، لن يكفيك هذا الكتاب، لأنَّه ليس مكتوبًا ليكون دليلًا شاملًا إلى اللغة. ومن هنا أحيلك إلى قراءة الكتاب الآتية لكي ترسِّخ فهمك للمعلومات التي أخذتها من هذا الكتاب، ولكي تستكشف وتتفحص مواضيع أخرى في JavaScript.

- JavaScript: The Good Parts, by Douglas Crockford
- JavaScript Patterns, by Stoyan Stefanov
- Object-Oriented JavaScript, by Stoyan Stefanov
- Professional JavaScript for Web Developers, by Nicholas C. Zakas
- High Performance JavaScript, by Nicholas C. Zakas